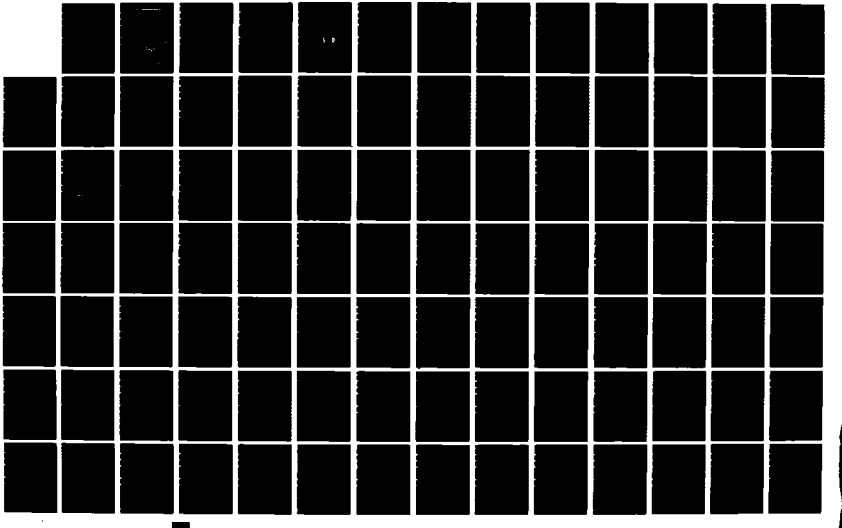
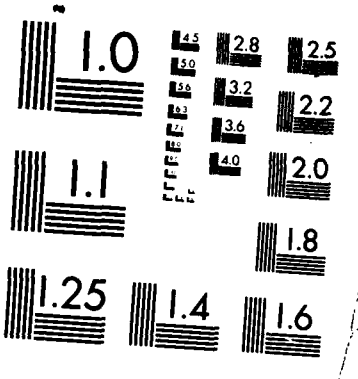
AD-A164 690 SYNTHESIS OF HIGH-SPEED DIGITAL SYSTEMS(U) UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES COMPUTER RESEARCH INST N PARK 08 NOV 85 CRI-85-23 ARO-20637.11-EL 1/3

UNCLASSIFIED DAAG29-83-K-0147 F/G 9/2 NL





(2)

Synthesis of High-Speed Digital Systems[1]

Technical Report CRI-85-23

Nohbyung Park

November 8, 1985

COMPUTER RESEARCH INSTITUTE

Department of Computer Science
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781
Telephone: (213) 743-3307

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADA164690

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ARO 20637.11-EL	2. GOVT ACCESSION NO.: N/A	3. RECIPIENT'S CATALOG NUMBER: N/A
4. TITLE (and Subtitle): Synthesis of High-Speed Digital Systems		5. TYPE OF REPORT & PERIOD COVERED: Technical, Jan. 84 - Oct. 85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Nohbyung Park		8. CONTRACT OR GRANT NUMBER(s): DAAG29-83-K-0147
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Southern California, Los Angeles, CA 90089		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709		12. REPORT DATE: November 1985
		13. NUMBER OF PAGES: 121
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automated design, synthesis, pipelining, overlapped execution, maximum execution overlap, clocking scheme synthesis, resource sharing, speed-cost tradeoff, scheduling, urgency measure, resource allocation, allocation table, latency, delay insertion.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design of high-speed systems and the automation of the design tasks involved is the topic of this thesis. In particular, the focus is on the use of pipelining and general overlapped execution. This thesis contains new techniques for speed-up, and describes the implementation of these techniques in a set of design automation programs. The new techniques produce designs

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

which share resources in a manner which is more complex than that used as common practice in working systems. The design of such complex hardware manually would be virtually impossible; thus, automated design is a key aspect of the proposed design methodology. The specific problem the thesis addresses is the following: the determination of the point in time each operation in a set of tasks is performed, and the manner in which the operators which execute each operation are reused from cycle to cycle. The first topic investigated in the thesis is clocking scheme synthesis. This type of synthesis involves selection of the number of clock phases, the duration of each phase, the amount of overlap in time between phases, and the assignment of functions to each clock phase. A theory of clocking is developed in the thesis, and used as the basis for software which automatically synthesizes clocking schemes. This program has been shown to improve manual designs by as much as a 60% performance gain. The potential for application of the clocking scheme synthesis to more general systems, including systolic arrays, is also described. Synthesis of pipelined data paths is the second topic investigated. This synthesis task involves the generation of data paths along with a clocking scheme which overlaps execution of multiple computation tasks. Theory of general execution overlap is presented, including a description of four different scheduling techniques. The third topic covered is the insertion of delays into pipelined systems to avoid resource conflicts while increasing performance. The thesis describes a set of algorithms which perform optimal delay insertion. The algorithms described in the thesis have been programmed in Franz LISP, and execute quickly for problems of practical size.

(2)

Synthesis of High-Speed Digital Systems[1]

Technical Report CRI-85-23

Nohbyung Park

November 8, 1985

---

[1]This research was supported by the Army Research Office (Grant DAAG29-80-k-0083 and Grant DAAG29-83-K-0147) and the International Business Machines Corporation (Grant S 956501 Q LX A B22).

SYNTHESIS OF
HIGH-SPEED DIGITAL SYSTEMS

by

Nohbyung Park

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

September 1985

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90089

This dissertation, written by

Nohbyung Park

under the direction of his Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

William D Jentzen
Dean of Graduate Studies

Date October 4, 1985

DISSERTATION COMMITTEE

Alice C. Parker
Chairperson

Mark H. Brenner

Gary R. Miller

DEDICATION

To

My father and mother,

and my family

ACKNOWLEDGEMENT

I thank my advisor Professor Alice Parker for her guidance, support and endless encouragement. Her enthusiasm towards research and her insight into the problems of design automation made this thesis possible. I will never forget her warm heart and friendship that she showed towards me even when she herself was struggling.

I thank Professor Melvin Breuer and Professor Gary Miller for their valuable criticisms and suggestions which enhanced both technical contents and readability of this thesis.

I thank my parents for their endless love, support and patience, which made it possible for me to continue my graduate study, and finally to finish this thesis.

My wife, Jung-Ju, and my two little angels, Sehwa and Julie, also deserve my sincere thanks for their patience and sacrifices throughout my graduate study.

I thank my friends John Granacki, David Knapp, Fadi Kurdahi and Sarma Sastry for the warm friendship. They were my English teachers, technical consultants, and buddies. Thank you guys for lending your hands and helping me throughout the struggle I had to fight for a small piece of truth.

I gratefully acknowledge the financial support received from the Army Research Office (Grant DAAG29-80-k-0083 and Grant DAAG29-83-K-0147) and the International Business Machines Corporation (Grant S 956501 Q LX A B22).

ABSTRACT

The design of high-speed systems and the automation of the design tasks involved is the topic of this thesis. In particular, the focus is on the use of pipelining and general overlapped execution in order to speed up the systems being implemented.

This thesis contains new techniques for speed-up, and describes the implementation of these techniques in a set of design automation programs. The new techniques produce designs which share resources and time in a manner which is more complex than that used as common practice in working systems. The manual design of such complex hardware would be virtually impossible; thus, automated design is a key aspect of the proposed design methodology.

The specific problem the thesis addresses is the following: the determination of the point in time each operation in a set of tasks is to be performed, and the manner in which the operators which execute each operation are reused from cycle to cycle.

The first topic investigated in the thesis is clocking scheme synthesis. This type of synthesis involves selection of the number of clock phases, the duration of each phase, the amount of overlap in time between phases, and the assignment of functions to each clock phase. A theory of clocking is developed in the thesis, and used as the basis for software which automatically synthesizes clocking schemes. This program has been shown to improve manual designs by as much as a 60% gain in performance. The potential for application of the clocking scheme synthesis to more general systems, including systolic arrays, is also described, and an example given.

Synthesis of pipelined data paths is the second topic investigated. This synthesis task involves the generation of data paths along with a clocking scheme which overlaps execution of multiple computation tasks. A theory of general execution overlap is presented, including a description of four different scheduling techniques. There is a brief description and example of the use of the pipeline synthesis software.

The third topic covered is the insertion of delays into pipelined systems to avoid resource conflicts while increasing performance. The thesis describes a set of algorithms which perform optimal delay insertion.

The algorithms described in the thesis have been programmed in Franz LISP, and execute quickly for problems of practical size.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1. The General Digital Design Optimization Problem

The general digital design problem is that of producing a hardware implementation of a system which exhibits a required behavior and satisfies any constraints imposed on it. Among the most typical design constraints are minimum required speed and maximum allowed cost and power consumption. Examples of desired goals are to maximize speed, to maximize speed-to-cost ratio, and to maximize speed-to-power consumption ratio.

Unfortunately, these optimization tasks often compete with each other. For example, the minimum cost implementation will rarely be the maximum speed implementation. For this reason, desired design goals are often used in addition to constraints in order to direct the optimization process towards a certain direction. Whenever there is more than one noninferior design alternative, the one that best meets the desired goals will be chosen.

The synthesis task can be partitioned into subtasks which will be repeated as the design proceeds towards a direction guided by constraints and desired goals, as listed below:

1. choose an appropriate design style (***design style selection***),

2. choose potentially optimal sets of functional and storage modules which can maximize speed and minimize cost and/or power consumption (***module selection***),

3. allocate operations and data values to functional and storage

elements. Partial interconnection may also be carried out (*resource allocation*),

4. find an optimal configuration and/or interconnection of modules so as to maximize performance. Detailed control hardware and/or microprograms are also synthesized during this phase (*configuration and interconnection*), and

5. for a given design which is non-optimal, find a near-optimal reconfiguration of the design within an allowed cost increase or speed decrease limit (*performance increase or cost reduction by reconfiguration*).

In cases when near-optimal solutions are desired, the complexity of these tasks is in decreasing order, since the solutions for the earlier phase problems can only be guaranteed to be optimal after a large number of (in worst case, all possible) solutions for the later-phase problems are compared. Unfortunately, finding optimal solutions even for some of the later phase problems is known to be intractable. For example, the resource allocation problem can be modeled as a *precedence constrained scheduling* problem, which is known to be NP-Complete [Gary 79]. Also, as a subproblem of subtask 4, the microcode compaction problem has been proven to be NP-Complete [Robertson 79]. Many other problems with exponential complexity in various design phases have been identified [Breuer 72, Sastry 82]. Only several problems of the last subtask turn out to be polynomial-time solvable [Dervos 83, Leiserson 83].

In practice, synthesis tasks are carried out by estimating and evaluating cost and speed of feasible hardware implementations of the system at various stages of the design process. Naturally, in order to carry out these tasks efficiently and to get a near-optimal design, a good estimation and evaluation strategy is crucial. Especially, in the last two phases described above, it is desirable that the speed estimation and evaluation procedures be able to suggest possible changes in the given design which can increase the speed.

1.2. Design Issues

1.2.1. Speeding up digital systems

Although there are many styles and variations of techniques for the control of high-performance digital systems, they can be classified as following two basic concepts:

- distributed processing under asynchronous distributed control, and
- overlapped (parallel) processing under centralized control.

The former class includes digital systems with multiple autonomous control sequencers such as multi-microprocessor systems, and VLSI circuits with multiple autonomous control modules and interfaces (e.g., a UART with separate sequencers for receiver and transmitter). The latter class includes systems or modules with only a single centralized controller. Any system belonging to the first class can be partitioned into subsystems and/or modules each of which can be classified under the latter class, although there are several complex control partitioning problems which must be addressed. The overall speed of such a distributed processing system will be determined as a function of the speed of each partitioned subsystem and/or module. Accordingly, we will focus our discussion on the latter case, overlapped processing under centralized control.

1.2.2. Logic design and clocking

In any sequential circuit, there are two logic components: the combinational logic blocks which perform the required functions and the storage elements which save the results of the combinational logic blocks. In the case of simple, sequential circuits, where an instruction or a state transition requires a part or all of the combinational circuitry and the results are latched into storage elements all at once, the interaction between the two components

is simple. The timing characteristics of the combinational circuitry dominates the determination of the speed of the circuit. However, in the case of complex, fast systems such as mainframe computer systems, a system is partitioned into subsystems (or stages as in a pipelined system) and fast multiple-phase clocks are used. System partitioning is performed for four reasons:

- due to physical limitations such as size or power consumption,
- to provide modularity and maintainability such as testing and design changes,
- to support resource sharing, and/or
- to support overlapped execution of different tasks.

In such a partitioned or staged system, the timing characteristics of the combinational and sequential components interact very closely and affect the performance of the systems in a different way. A computation task is partitioned into subtasks and subtasks are carried out in many subsystems or stages. The results of one or more subsystems may need to be fed to other subsystems to complete the original task. Then, the subsystems freed up can be used for other tasks. There are two obvious objectives in designing such complex systems:

- optimal system partitioning into subsystems, and
- optimal sequencing control to carry out tasks and subtasks.

Both partitioning and sequencing control require proper buffering and synchronization of value transfer from subsystem to subsystem, which involves placement of storage elements in between and/or inside each subsystem, and the proper timing and sequence for clocking the storage elements.

1.2.3. An example digital circuit design task

In any digital system, cost (sometimes analogous to chip area) and speed are traded off to get a system which meets particular requirements. How cost and speed are traded off, however, varies from design to design. The complexities of the problem are illustrated by a brief multiplier example.

Suppose we have decomposed a 16-bit multiply operation as shown in Figure 1.2-1, using 8-bit multiply and addition operations. The nodes indicate operations, and the arcs indicate values. The task is to design an inexpensive, fast multiplier from this data flow graph. An obvious solution using four 8-bit multipliers (M_1 through M_4) and five 8-bit adders (A_1 through A_5) is shown in Figure 1.2-2. The timing diagram for this multiplier is shown in Figure 1.2-3, where the delay time of an 8-bit multiplier and an 8-bit adder are 80 and 30 nsec., respectively. The register propagation delay is 10 nsec.

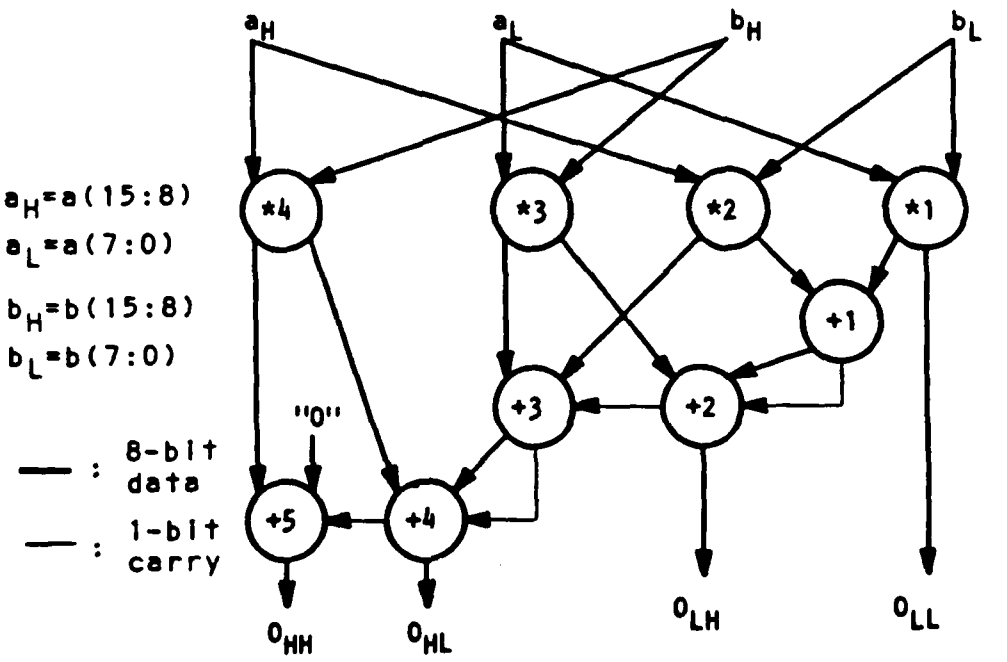


Figure 1.2-1: A data flow graph for a 16-bit unsigned integer multiplication.

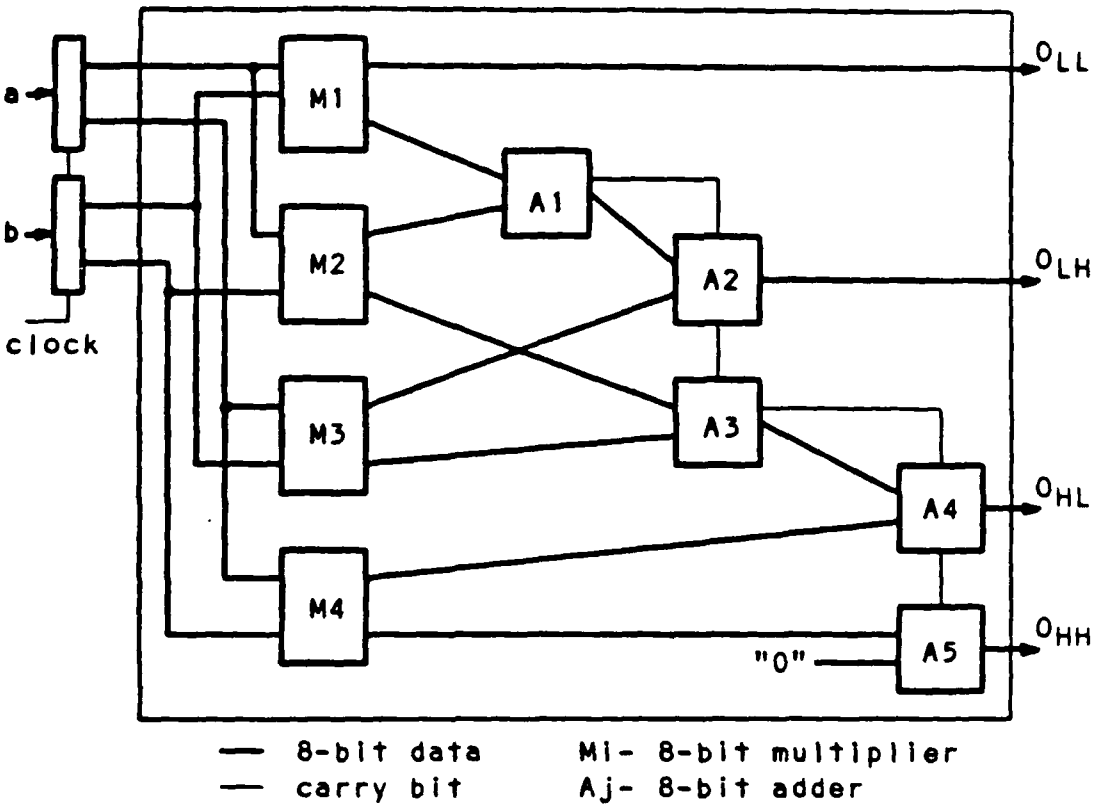


Figure 1.2-2: A 16-bit parallel unsigned integer multiplier.

However, since there has been no reuse of resources, the resources are only busy a fraction of the time. There are two possible ways to increase resource utilization without sacrificing performance.[1] These are

1. use less modules and reuse them for different operations as many times as necessary, or

2. when multiple tasks are to be executed sequentially, partition each task into subtasks and execute subtasks of multiple tasks on different parts of the circuit simultaneously (overlapped execution of multiple tasks).

---

[1]We could increase resource utilization greatly by using only one multiplier and one adder, but at a great sacrifice in performance!

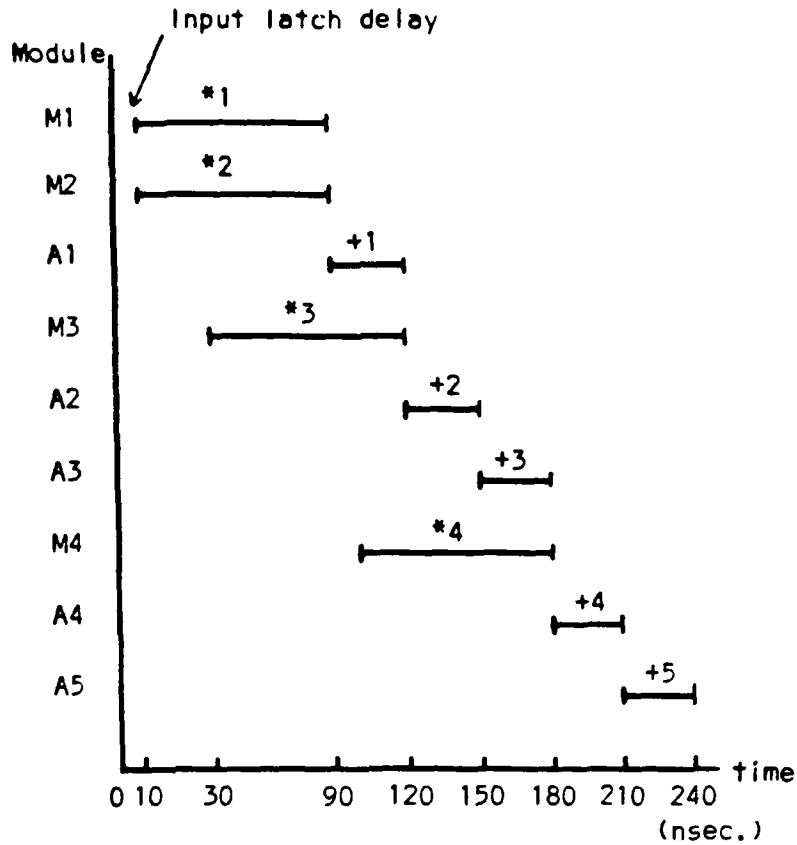


Figure 1.2-3: Timing diagram for the multiplier of Figure 1.2-2.

The first method takes a local view of the multiply operation, and assumes it is required to be performed only as often as the outputs can be produced. No other multiply operation would require the hardware until the present task was completed. As long as multiplies come along slower than the speed of the multiplier, this is a valid viewpoint.

Sometimes, we cannot consider a single multiply operation in isolation. In some cases the multiply tasks can be generated at a faster rate than the total multiply time. In this case, we must begin a second multiply before the first is completed, utilizing the resources left idle by the first multiply as it goes into the later stages of its computation. In the extreme case, we can overlap

enough multiply operations so that the resources are all busy all the time, and the multiplies are handled as fast as required. This *overlapping in time*, in its simplest form, is sometimes referred to as **pipelining**.

An implementation of the multiplier with overlaps is shown in Figure 1.2-4. This multiplier uses the same set of functional modules as the multiplier in Figure 1.2-2. Only some latches have been added. These latches divide the circuit into two stages and prevent direct flow of values between stages. For this reason, they are called **stage latches**.

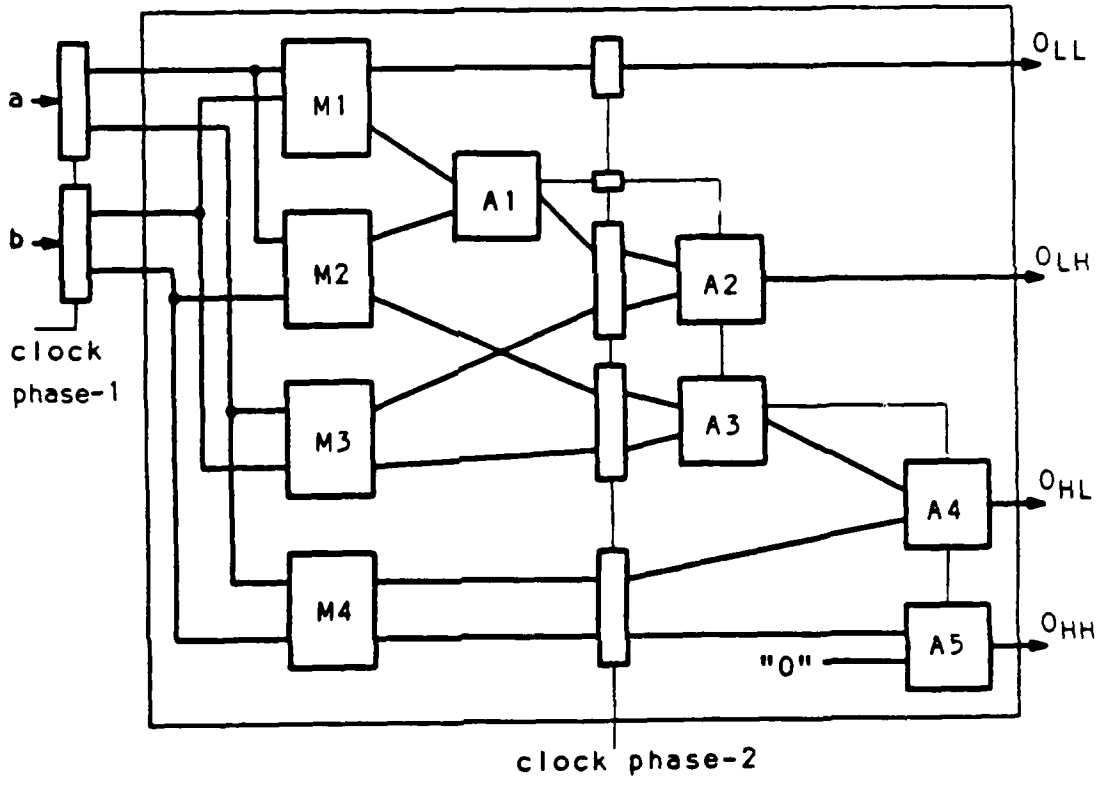


Figure 1.2-4: A multiplier with execution overlap.

With this multiplier, a 16-bit multiply is performed in two steps as follows. (Refer to the data flow graph of Figure 1.2-1.)

1. The input latches are clocked by clock phase-1 and store the inputs. 8-bit multiplications *1 through *4 are performed by 8-bit multipliers M_1 through M_4. As soon as the results of *1 and *4 are available, the +1 operation is performed by adder A1.

2. The stage latches are clocked by clock phase-2 and store the results of the previous step, i.e., the outputs of M1 through M4, and A1. Addition operations +2 through +4 are performed by adders A_2 through A_5 and the results are output.

The timing diagram for this multiplier is the same as Figure 1.2-3 except that the operations +2 through +5 are delayed by 10 nsec. due to the propagation delay of the stage latches. Therefore we cannot perform a multiply operation faster than before. However, the effective throughput is greatly increased since we can perform more than one operation in parallel. *As soon as a 16-bit multiply operation moves to the second stage by storing the results of the first stage in the stage latch, another 16-bit multiply can start on the first stage.* On this multiplier, a new multiply task can be initiated every 130 nsec. The resource utilization for the 8-bit multipliers has increased to 61 percent as a result of the overlaps.

Figure 1.2-5 shows the timing of overlapped execution of three multiply tasks, I_1, I_2, and I_3, using an overlapped **multi-phase** (two-phase) clock which we will discuss in the following section.

The choice of the two speed-up techniques depends on the design constraints, the desired optimization goals and the input task stream. If the computation tasks arrive at a high rate and the designer wants to make the design as fast as possible, he may choose the latter technique. On the other hand, if the cost budget (either the total cost or the total area) is very tight, he would choose the former technique. Depending on the tightness of the constraints, the desired optimization goals, and the arrival rate of the tasks, the best choice might be a mixture of both techniques.

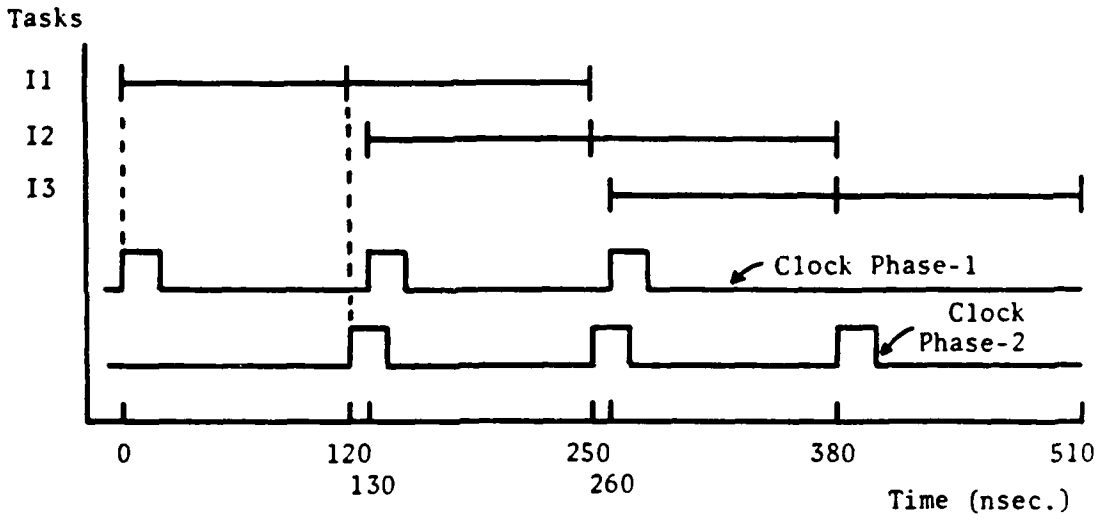


Figure 1.2-5: Overlapped execution on the multiplier of Figure 1.2-4.

1.2.4. Overlapping multi-phase clocks

A multi-phase clock cycle consists of a sequence of clock phases delayed from a single clock source. Since each phase is delayed from a single clock source, all the phases have the same pitch, which is called a **clock period**. The time interval from the tick of a phase to the tick of its next phase is called the **length** of the phase. Each phase can be of different length. An example of a non-overlapping 3-phase clock is shown in Figure 1.2-6. In case of a non-overlapping multiphase clock, the clock period is the same as the sum of the lengths of the clock phases.

An overlapping multiphase example is shown in Figure 1.2-7. In the case of an overlapping multi-phase clock, the clock period is shorter than the total sum of the clock phases. A new clock cycle starts before the last clock phase of the previous cycle.

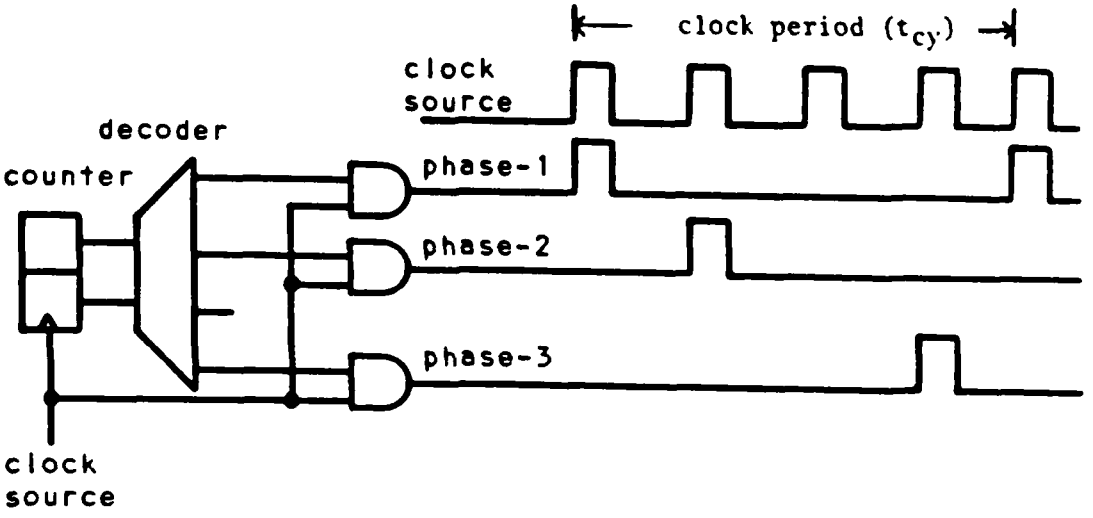


Figure 1.2-6: A non-overlapping 3-phase clock.

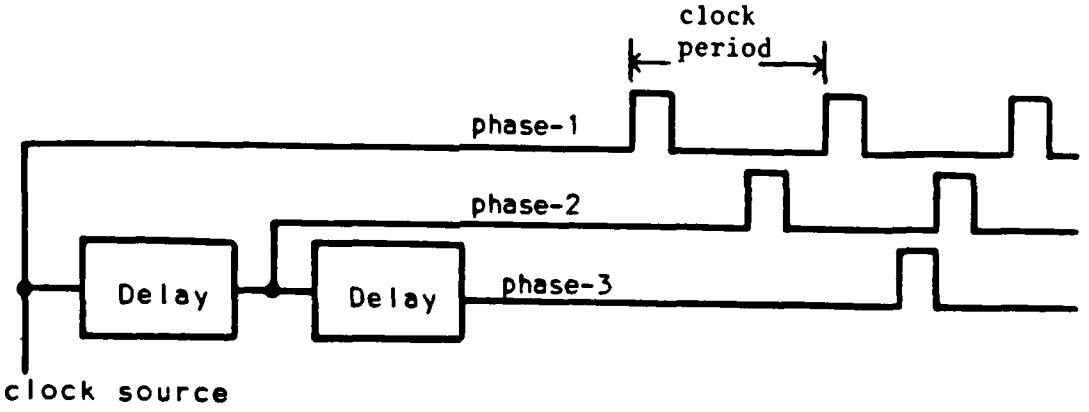


Figure 1.2-7: An overlapping 3-phase clock.

The clock needed for the sequencing of Figure 1.2-5 is an overlapping two-phase clock with a clock period of 130 nsec. The length of the first phase is 120 nsec. and the second phase 130 nsec.

1.2.5. Two sequencing levels of a digital system

In digital systems with two-level control structures, sequencing is carried out on two levels, the **macro** and **micro** levels. An execution instance of a machine instruction or a major loop of a finite-state-machine (**macro task**) corresponds to a **macro cycle** and an execution instance of a microinstruction or a state of an finite-state-machine (**micro task**) corresponds to a **micro cycle**, which are carried out by a **macro engine** and a **micro engine**, respectively. Most Von Neumann-type computer CPUs and simple digital systems have a two-level control structure. In most digital systems whose control structure has more than two levels, we can also find similar levels of sequencing corresponding to the macro and micro levels.[2] Figure 1.2-8 shows an example of a microprogrammed computer CPU.

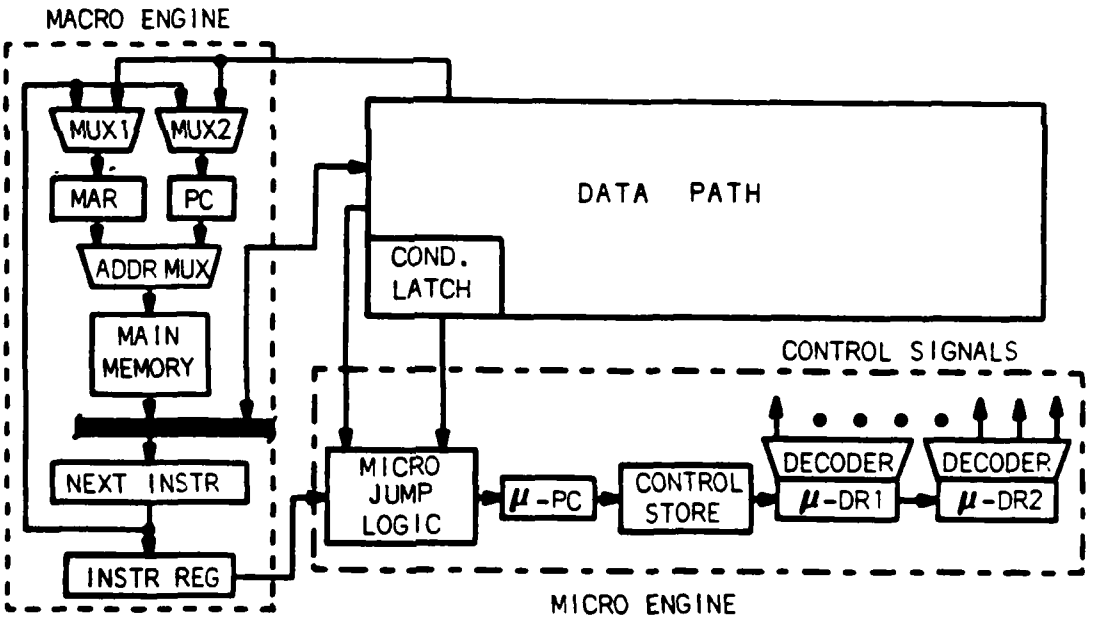


Figure 1.2-8: Sequencing engines of a digital system.

---

[2]For a nano-programmed CPU such as the Nanodata QM-1, nanoinstruction cycles can be considered to be the micro cycles, and microinstruction cycles and machine instruction cycles can be merged and considered to be the macro cycles of our classification.

Macro cycles consist of sequences of one or more micro cycles. Overlapping macro cycles are implemented by proper partitioning of macro cycles into sequences of micro cycles. *For example, an operand needed by the current machine instruction being executed could be fetched by some microinstruction of the previous machine instruction and some microinstruction of the current machine instruction may fetch the next machine instruction in advance.*

A micro cycle consists of **minor** cycles (register-transfers). Each minor cycle reads from storage elements, transforms and stores data and/or control values to storage elements which are used to buffer the flow of the values between functional elements.[3] Such buffering storage elements are called **stage latches**. For the micro engine of figure 1.2-8, μ-PC, μ-DR1, μ-DR2 and "Cond. Latch" can be considered to be stage latches. In general, any storage element in the system can be a stage latch.

1.2.6. Overlapped execution in micro-level sequencing

At the macro level, various techniques for speeding up digital systems exist. Examples are instruction look-ahead [Keller 75], stack architectures [McKeeman 75], and dataflow machines [Dennis 75]. However, the ultimate performance of these speed-up techniques will depend greatly on good sequencing control schemes at the micro level, since each macro-level task is eventually implemented by micro cycles.

At the micro level, execution overlap is achieved by overlapping the execution of minor cycles of multiple micro cycles. As shown in Figure 1.2-9-(a), simple overlap is often used in small computer CPUs. As data path

---

[3] In other words, a minor cycle corresponds to a primitive operation at the register-transfer level.

cycles overlap ((b) and (c) of Figure 1.2-9), overlap within functional units can also be used (e.g., the pipelined multiplier of the IBM 360/91). Possible places where micro-level overlap can be achieved are

1. between stages of the micro engine,
2. between the micro engine and the data path, and
3. between the data path stages.

Figure 1.2-9 shows timing examples of micro cycles. Case (b) corresponds to the digital system of Figure 1.2-8, where, for a microinstruction i, the minor cycles Ii1, Ii2, Ii3 and Ii4 start by clocking μ-PC, μ-DR1, μ-DR2 and "Cond. Latch", respectively. As we show later, if there is no conflict in stage usage and no branches are executed, the maximum execution speed of a micro engine is determined by the longest interstage propagation delay (which is the minimum possible clock period) as in static pipelines without loops. Of course, the actual interstage propagation times depend on the number and length of the clock phases.

As shown in Figure 1.2-9, a branch delays fetch of the next micro cycle until the earliest fetch clock phase after the completion of the branch. Accordingly, the time delay due to a branch is a function of the execution time of the branch and the clock period. Thus, the overall performance of the micro engine also depends on the sequence of micro cycles to be executed.

Resource conflict and data dependency (data precedence) are other factors reducing the advantages of execution overlap. For example, two microinstructions, I_i and I_{i+1}, are executed consecutively and each has three microoperations (data path cycles), as follows:

Iij : j-th minor cycle of micro cycle i.

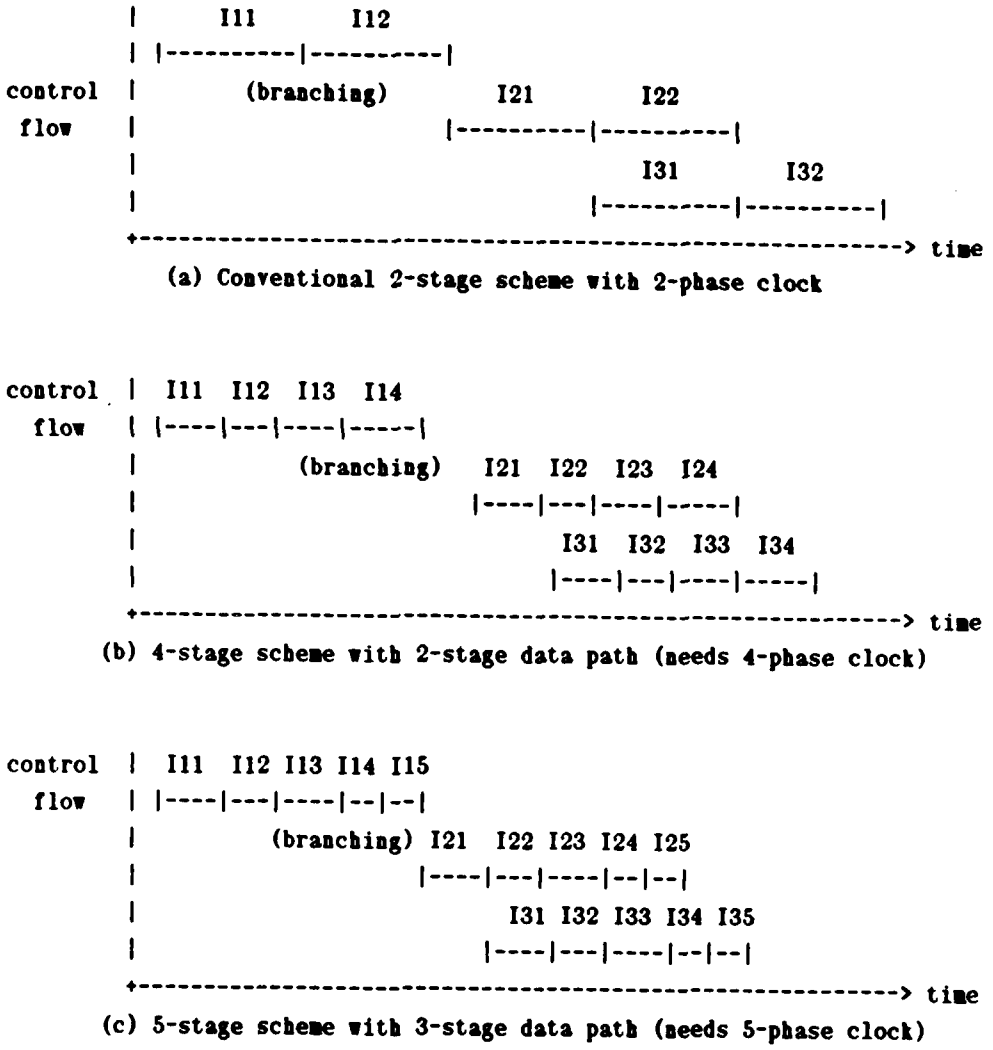


Figure 1.2-9: Examples of micro cycle sequencing (Gantt Chart).

	I_i			I_{i+1}	
$I_{i,1}$:	C <-- MDR	NEXT	$I_{i+1,1}$:	C <-- D*2	NEXT
$I_{i,2}$:	A <-- B + C	NEXT	$I_{i+1,2}$:	C <-- C + 2	NEXT
$I_{i,3}$:	A <-- A*2		$I_{i+1,3}$:	E <-- C + D	

$I_{i+1,1}$ has a data-dependency relation with $I_{i,1}$ and $I_{i,2}$ [Dasgupta 76]. It also has a resource conflict (assuming only one multiplier) with $I_{i,3}$. Thus $I_{i+1,1}$

cannot start execution until all the data path cycles of I_i complete. These situations can also be found in pipelines where a stage is used more than once during a computation of a single task [Kogge 81]. Forbidden latencies of a pipeline are determined by resource conflicts between tasks and loops are major contributors to them. These problems slow down the execution speed as well as increase the complexity of the control circuitry.

The delay of an initiation of a task due to any of these problems is called **resynchronization** of the pipeline, and the time overhead due to such a delay is called **resynchronization overhead**.

Similar problems arise at various levels of most digital designs [Ramamoorthy 77, Kartashev 82]. The higher the level, the harder the analysis and the higher the control cost. An example of a higher-level resynchronization is a page fault in a virtual memory computer system. If the page fault was for a new instruction, the pipeline must be stopped until the needed page is read in from the disk, which usually requires quite a long time compared to the speed of the pipeline.

At the micro-sequencing level, these problems can be analyzed in a formal way using a graph-theoretic, algebraic methodology, which will be discussed in this thesis.

1.2.7. Definitions of the speed of digital systems

As mentioned in the previous sections, the system tasks (processes or programs) consist of sequences of macro tasks, each of which consists of one or more fixed sequences of micro cycles. Therefore, execution times of system tasks can be determined by the execution sequences of their micro cycles and the execution time of those sequences. In this sense, the execution speed of the micro engine can represent the execution speed of the system. There are

several ways of defining the execution speed of a micro engine of a digital system for performance evaluation:

1. maximum possible execution speed,

2. execution speed for certain micro cycles, and

3. execution speed for a (weighted) average mixture of micro cycles.

The first two parameters are not realistic since they do not involve an actual mixture of the micro cycles. The third parameter, which is an overall performance measure of the system, can be computed by assuming the average mixture of micro cycles over a long enough time period. The total estimated execution time divided by the number of micro cycles executed will be the average expected execution time of a micro cycle. Appropriate weighting functions may be used to indicate the average occurrence and/or importance of each micro cycle.

1.3. Research Overview

In this thesis, we consider synthesizing high-performance digital systems with centralized control at the micro level. The main objective of the research is to achieve maximum performance with minimum hardware cost and design effort.

Among the most costly and time consuming tasks at the early stages of data path design are module selection and allocation, which select functional and storage elements and assign functions and values to them. Also, during or after the module selection and allocation phases, control is synthesized, involving the synthesis of either a microprogrammed or a hardwired sequential machine. When near optimal design is required, all these tasks are computationally intractable.

Furthermore, once all these tasks are completed, any non-trivial change in either control flow or data flow may require almost the same effort as the initial design. Naturally, we consider the following two fundamental questions:

1. During the module selection and allocation phases of the data path design (assuming a fixed control sequence), how can we efficiently estimate and compare the performance of alternative decisions?

2. For a completed design, how can we increase the performance of the system at minimum hardware cost and minimum design and/or design change time?

The main goal of this research is to develop a methodology which can answer these questions at the micro level of digital systems.

We divide the design tasks into two cases depending on the design input and design constraints as follows:

1. Case 1: The maximum possible performance is desired with no cost constraint.

2. Case 2: A cost/performance tradeoff is either forced due to the cost constraint or desired for cost and speed optimization.

1.3.1. Clocking scheme synthesis for maximum execution overlap

In the first case of the design tasks mentioned above, we assume there is no resource sharing within a micro cycle in order to maximize the performance, i.e., no micro cycle is using the same module more than once. Let us consider this case now.

As mentioned in Section 1.1, the module selection and allocation phases of the design are complicated, timing consuming tasks. Also, whenever there is any change in either module selection or allocation, both the data and control flow must be altered to take maximum advantage of the change. Thus, the task will involve almost the same amount of work as the initial design. For example, if a new, faster ALU is chosen, it often requires rewriting the microprogram as well as changing the values allocated to the operand and/or result registers. This automatically involves changing the interconnections for both the data flow and control flow.

In order to avoid such costly and time consuming iterations for this task, we consider adding or reconfiguring only storage elements. This can be done without altering the basic structure of the original control and data flow and thus is considered to be transparent to the data flow and control flow analyzer. This gives the designer an efficient estimation tool for the comparison of alternative module selections and allocations.

Assuming that the control sequence of the micro cycles is fixed, we consider two basic approaches to the above problem:

1. For a set of allocated functional modules (for both the data path and the micro engine), add and connect the minimum number of storage elements necessary to achieve a certain level of performance.

2. For a completed design, add a certain number of storage elements to the data path and/or micro engine in such a way that the performance increase will be **maximized** by virtue of maximum execution overlap of the micro cycles.

In either case, we try to maximize execution overlap of the micro cycles considering the time overhead due to branches, resource conflicts and data dependency relations. Maximum execution overlap can be achieved by synthesizing an optimal clocking scheme, which involves the following tasks:

- optimal assignment, relocation, addition or deletion of the stage latches,

- choice of an optimal clock period and the number and lengths of the clock phases, and

- optimal clock signal gating and routing.

In carrying out these tasks, we formulate the problem as a graph theoretic problem. Digital circuits are modeled by directed graphs which show the pathways of the data and control flow. By properly weighting the vertices and the directed edges, we can model the execution sequences of the micro cycles as tours on the graphs. Also, the time taken by each segment of the tours can be computed easily. Assigning and/or inserting stage latches can be modeled as finding multiple-edge cut sets. Once the locations of the stage latches are determined, then the optimal clock period and clock sequence can be computed, considering the synchronization overhead discussed before. Software has been written to perform these tasks.

1.3.2. Pipeline synthesis

For the second type of design task described above, we must consider sharing resources, even during a micro cycle, in order to meet certain cost limits. Due to the control complexity of general execution overlap with resource sharing,[4] we make an assumption that a micro cycle is divided into a sequence of equal length minor cycles, which reduces the design problem to a pipeline style.

In pipelining, each unit computation task (corresponding to a micro task) is partitioned into a sequence of subtasks (corresponding to minor cycles) and each subtask is assigned to and executed during a phase or a clock cycle. Every phase has the same time period. Consecutive tasks are initiated at some fixed or variable intervals which are integer multiples of a phase and shorter than the execution time of a unit task. In this fashion, subtasks of consecutive tasks are to be executed in parallel and, in fact, consecutive tasks are to be executed overlapped in time on different parts of a circuit.

We approach the problem of pipeline synthesis from the behavioral level of design description. A data flow graph model is used to specify behavior. It can have both

- multiple disconnected graphs for parallel independent tasks, and
- conditional branches allowing modeling of more than one task in a single data flow graph.

The design inputs are the data flow graph, specifying the desired behavior of the pipeline, and design constraints on either cost or speed. We assume that module selection is done, i.e., we know that which operations or

---

[4]This problem is considered to be a combination of resource-constrained scheduling and predecence constrained scheduling problems, each of which is NP-Complete [Gary 79].

types of operations are to be executed by which type of modules. The scheduling task is modeled as a graph partitioning into disconnected subgraphs, where a partitioned subgraph corresponds to a time step of the pipeline execution. The distribution of the operations in the subgraphs determines the number of functional modules and multiplexers needed. The locations of the edges separating subgraphs determine the number and locations of stage latches. The critical path of a stage determines the clock cycle of the pipeline. The number of subgraphs corresponds to the number of clock cycles for a task to complete.

The goal of the scheduling task is either

1. maximizing the effective performance while satisfying the cost constraint, or

2. minimizing the total cost while satisfying the minimum performance constraint.

For a sequence of tasks, the maximum performance of a pipeline is determined by

1. the minimum possible clock cycle time,

2. the minimum possible initiation interval between consecutive tasks, and

3. the number of clock cycles required to complete a task.

However, optimization tasks for these three performance parameters compete with each other. For example, partitioning the data flow graph into a smaller number of subgraphs will increase the length of the critical path in each subgraph. Due to resource sharing, longer subgraphs will have more possibility of resource conflict when they overlap. However, making the subgraphs short will increase the number of stages in the pipeline, which increases the number and cost of stage latches, as well as slows down the

execution time of a task due to the propagation delays of more stage latches. In either case, scheduling and resource allocation are constrained by either the cost or the performance constraint.

In this thesis, we present a set of fast scheduling procedures which are iterated and guided by a good optimization strategy. These procedures carry out resource allocation at the same time in order to make the schedule as feasible as possible and to maximize the performance. After scheduling is done, the actual cost is estimated by adding the cost for latches and multiplexers. The next scheduling cycle is guided by the performance and cost estimation for the previous schedule.

These procedures have been programmed and tested. Test results show that these heuristic procedures produce near-optimal schedules in most cases. These heuristic procedures have polynomial time complexity ($O[n^2 log n]$) and will be an extremely useful tool for quick design space exploration.

1.3.3. Exhaustive algorithms

We also examine the possibility of using an exhaustive algorithm for an optimal scheduling of the operations while satisfying the cost constraint. This exhaustive scheduling algorithm takes the best result of polynomial-time scheduling algorithms as a good lower bound on performance to prune the search space.

Finally, we discuss another exhaustive algorithm for performance improvement of an already existing static pipeline,[5] by inserting non-operational delay cycles into some stages into the pipeline.

---

[5] A pipeline without any conditional branches.

1.4. Related Work

Since the task of high-level (functional level) digital design synthesis was launched more than a decade ago [Friedman 69, Darringer 69], there has been a vast amount of effort in automating various steps of the digital design process such as design style selection [Thomas 77, Lawson 78], data path synthesis [Irani 72, Snow 78, Hafer 81, Hitchcock 83, Kowalski 83, Girczyc 84, Tseng 84], microprogram synthesis [Agerwala 76, Patterson 76, Nagle 80], and integrated design automation systems [Parker 79, Estrin 78, Zimmermann 79, Foulk 80]. However, there has not been much work in the area of clocking scheme synthesis except that done for pipeline control [Cotten 65, Davidson 71, Davidson 75, Patel 76, Kogge 81] which can be considered to be a special case of general clocking scheme synthesis. As we have discussed before, the clocking scheme synthesis task is important in optimizing the speed of digital systems and must be carried out together with the data path and control design. However, it has been either buried under architectural design [Parker 79, Estrin 78, Zimmermann 79, Foulk 80], or assumed *a priori*, as a part of the control design [Boulaye 71, Katzan 71, Nagle 80, Andrews 80] or data path design [Hafer 81]. In some cases, clocking scheme synthesis alone is carried out for already completed designs [Davidson 75, Leiserson 82, Leiserson 83]. Among them, we will briefly discuss several which are most closely related to our research.

1.4.1. Clocking scheme synthesis

Leiserson [Leiserson 82, Leiserson 83] proposed a technique which determines a relocation of the registers in a given systolic-type data path in order to minimize the clock period. The data path is modeled as an acyclic directed graph where the vertices represent functional modules and the directed edges represent interconnections. The locations and the number of registers are indicated by the edge-weights. The basic assumption of this

technique is that all the hardware modules are performing useful operations at any time and thus all the registers are clocked at the same time by a single clock source (e.g., a systolic array). The technique moves the registers of the original design along the direction of the data flow. If the movement is to be made onto any forked[6] edges, registers are copied to all of the fork-edges in order not to change the original data flow. Figure 1.4-1 shows an example of step-by-step relocation of registers.

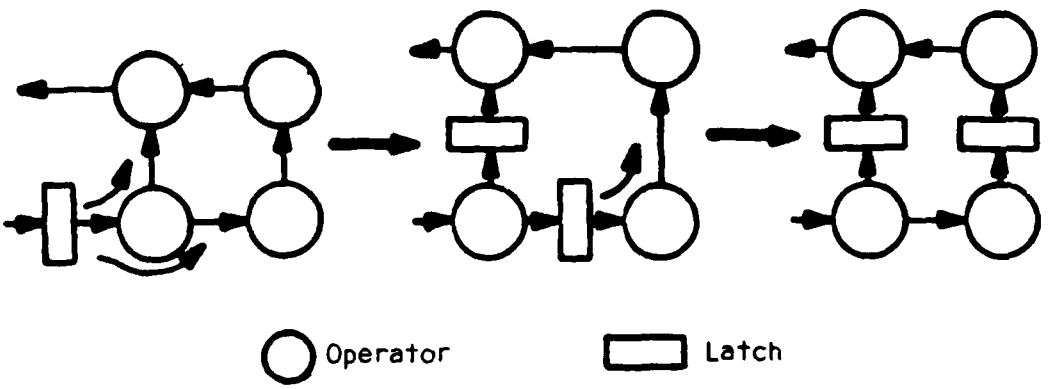


Figure 1.4-1: Leiserson's retiming.

The optimal relocation of registers is determined as the design in which the longest propagation delay between any two registers is minimized, which minimizes the clock period. The major contribution of this work is that it suggests several formal tools for timing analysis of digital circuits, which are a graph model of the digital circuit and a problem formulation using linear and/or mixed-integer linear programming. There are several shortcomings of this technique to be used in the general case of digital systems speed-up. They are

- the technique assumes fixed clocking for all the registers at the same time,

[6] In this model, all the forks are AND forks since every hardware module is performing useful operations and thus all the interconnections are carrying useful data values.

- the technique assumes fixed data flow for all time,
- control hardware timing is not considered, and
- register propagation delays are ignored.

Also, this technique does not handle execution overlap between multiple computation tasks. In Chapter 4, we will show an example of how the clock period can be reduced further by using execution overlap.

Boulaye [Boulaye 71] discusses speeding up pipelined micro engines by minimizing the time overhead caused by the conditional branches, which is accomplished by clocking the condition latch (which contains values that determine selection of the next microinstruction) as early as possible. This approach can also be considered as relocation of registers to reduce the critical path or the critical stage of a pipeline. However, if the propagation delays of both the data path stages and the stages of the micro engine are not considered together, optimal relocation cannot be determined. Also, in an instruction pipeline, branches cause resynchronization overhead, which also involves the termination and re-initiation of the data path stages.

Andrews [Andrews 80] considers using multiphase clocking as one way of reducing the number of microinstruction fetches from slow microcode storage. By using a multiphase clock, microinstructions can be horizontally coded and executed serially in several clock phases without having an expensive data path. As he mentions, the performance of this technique depends on the coding efficiency of the horizontal microprogram. If the microinstructions are sparsely coded, then the resource utilization efficiency will be low. Also, after the completion of a microinstruction which has only microoperations with short execution times, there will be idle time until the fetch cycle of the next microinstruction. This is true for any microprogram (vertical or horizontal) if execution overlap is not used. However, if execution overlap is used, this is not

always true, since the execution speed of the micro engine depends much more on the longest microoperation execution time rather than the total execution time span of each of the microinstructions. Moreover, as we shall show in Chapter 3, if execution overlap is properly used, vertically coded microinstructions can be executed as fast as horizontally coded ones. This saves a significant amount of design time and avoids the complexity of horizontal microprogram compaction, which is known to be intractable [Horowitz 78, Robertson 79].

As another approach, Berg [Berg 79] characterized the timing behavior of a given control and clocking scheme in order to provide a guideline for the synthesis of a fast and correct microprogram. The timing behavior of a controller at the macro level is modeled as a finite state machine. The model allows multiphase execution of micro-instructions. However, the model is focused on timing the interactions between main system blocks such as the CPU, main memory, and I/O controller.

1.4.2. Pipeline clocking schemes

Basic timing requirements and clock skew problems for pipelines were first analyzed by [Cotten 65], and well explained in [Kogge 81]. If the shortest path through a stage is shorter than the width of a clock pulse, then the output of a latch can change the input of the next stage latch while the clock pulse is active, which is termed a *critical race* by [Cotten 65]. There are four solutions to the critical race problem:

1. the short paths which might cause critical races must be augmented by time-delaying and doing-nothing circuits [Cotten 65],

2. the width of the clock pulse must be reduced by using a faster latch,

3. master-slave flip-flops must be used, or

4. edge-triggered flip-flops must be used.

A recent work by [Unger 83] at IBM discusses the third and forth solution approaches. In brief, master-slave and edge-triggered types of latches/flip-flops are race-free themselves and therefore can prevent critical races.

Another problem, called *clock skew*, must be considered in order to get the right timing, and cannot be avoided even with master-slave or edge-triggered flip-flops. In real designs of a pipeline, stages are physically apart from each other. Different interconnection lines carrying the clock signal have different lengths and loading effects causing different transmission delays. Also, clock signal drivers have different delay times depending on the size. Due to all these factors, the arrival times of the clock signal to different stages are not the same. This may cause latching wrong results by clocking latches either too early or too late. The maximum difference of any two clock signal arrival times is called *clock skew*. A solution to the critical race problem due to clock skew has been proposed by [Cotten 65]. It uses a chain of clock pulses to clock the latches in sequence instead of clocking all the latches at the same time with a single clock source.

All the timing issues we have discussed in this section are critical to pipeline design at the gate level, where delay times of the latches are comparable to the stage logic block delays. Although the basic problems remain the same, however, in the case of pipelined designs at the functional module (e.g. a 4-bit adder, an ALU, etc.) level or register-transfer level, these issues have relatively little importance compared to partitioning the circuit into stages and scheduling computation tasks on the pipeline.

1.4.3. Pipeline scheduling

The pipeline scheduling problem has been studied by several researchers. In general, most scheduling problems (e.g. job shop scheduling, flow shop scheduling, etc.) have been known to be NP-Complete or NP-hard [Baker 74, Horowitz 78, Gary 79]. Ramamoorthy and Li [Ramamoorthy 75] have shown that the general pipeline scheduling problem is also NP-Complete. Some engineering solution techniques for pipeline scheduling have been developed in a series of papers, [Davidson 71, Shar 72, Davidson 75, Patel 76]. We summarize the common characteristics of these techniques.

The basic tool for schedule analysis is the reservation table developed by Davidson [Davidson 71]. A reservation table is a two-dimensional tabular description of the pattern and sequence of stage usage of a computation task. Each column corresponds to a time step and each row corresponds to a stage. Resource conflicts or *collisions* between any two consecutive computation tasks can be detected by superposing their reservations. The goal of the scheduling task is to find out an optimal sequence of initiation intervals for a sequence of computation tasks such that the average initiation interval is minimized.

The lower bound on the average initiation interval for a reservation was computed by Shar [Shar 72], and is equal to the maximum number of marks in a single row in terms of the number of time steps. However, there is no guarantee that the lower bound can be achieved, depending on the pattern of the reservation table. The reason is that the desirable small initiation intervals may not always be permissible due to collision. Another situation that possibly makes the average initiation interval longer than the lower bound is when an already existing pipeline is asked to perform a new type of computation. The new type of task may have very much different stage usage, which may increase the average initiation interval. In such a case the designer has little

control over the reservation table of the new type of task. A solution technique to these problems has been reported by Patel [Patel 76]. The solution technique selectively delays some of the marks in each row of the reservation table. The number of marks in each row is not changed. One column delay of a mark corresponds to a delay of a time step. In brief, the marks in each row is relocated such that a certain desired sequence of initiation intervals does not cause any collision.

The actual implementation of a delay step can be either (i) a real empty stage with only stage latches or (ii) skipping clock phases, depending on whether buffering (saving) the inputs for the delayed computation is necessary. In other words, if the latches storing the input values for a delayed computation are not overwritten by some other computation step, there is no need to pipe the values through any other storage elements.

All the pipeline scheduling techniques we have discussed in this section assume the following three restrictions:

- Partial sharing of a stage (substaging of a stage) is not allowed.
- The execution time for all stages is fixed and the same.
- No conditional selection of stages is allowed during the execution of any computation (i.e. the sequence of stage usage of any task is always fixed).

These restrictions constitute the major differences between the general execution overlap problem we are studying and the traditional pipelining problem discussed above. The third restriction, especially, prevents the usage of these synthesized pipelines in general-purpose digital systems such as a computer CPU or a digital controller. Also, the first restriction makes the cost-speed tradeoff of pipeline designs rigid and leaves little freedom. In Chapters 4 and 5, we will discuss how we can remove the first and third restrictions.

1.4.4. Other related work

The basic concept of execution overlap at the micro-level under a centralized control originates from the look-ahead technique [Keller 75] for the prefetch of instructions and operands. Examples of machines which implement macro level execution overlap include the CDC6600, and the IBM 360/91, 195 and 370/165. They assume that instruction fetch, decode, and execute cycles, each consisting of a sequence of micro cycles, take almost equal time, which is the basic assumption of general pipelines. Possible execution overlap is predicted by checking the type and execution status of the current macro task being executed. Typical checking mechanisms use condition flags and/or counters which represent the state of associated resources. Naturally, look-ahead techniques assume flexible execution control mechanisms implemented by micro-level sequencing primitives [Katzan 71]. However, at the micro level, implementing look-ahead is costly and difficult since the look-ahead mechanism must be much faster than the micro cycle time in order to achieve execution overlap, which, in most cases, requires hardware level primitives.

Nagle [Nagle 80] provides a good insight into the general problems of control synthesis at the micro level, although all the problems are not analyzed in depth. The major contribution of this work is microprogram synthesis under given constraints, such as the capacity of the microprogram storage, speed requirements, and the number of control signals that can be activated at the same time. The control-flow optimization and control distribution techniques proposed can be used to reduce the number of branches, to shorten conditional branching time and to reduce the number of micro cycles, which is essential to increase the performance of the micro engine.

Cook [Cook 79] considered multiphase clocking of PLA's in order to reduce the power consumption of the PLA. A precharge scheme using a multiphase clock is used to compensate the turn-on/off time delay. Although

he does not mention it, his PLA partitioning technique may be very useful for multistaging a control store using PLA's. For example, we can partition the AND-plane and OR-plane of a large PLA by inserting a latch to store the product terms. Then, since it is a sequential machine design, we can overlap the propagation delays of the AND-plane and the OR-plane.

A technique for performance measurement of static pipelines is proposed by Lang [Lang 79], in which a pipeline is modeled as chains of processing elements and a table-driven simulation method is used for the performance estimation.

1.5. Thesis Outline

Chapter 2 describes the theory of maximum execution overlap. The clocking scheme synthesis task is defined, and the problem of optimal clocking scheme synthesis for maximum execution overlap is formulated in graph-theoretic terms. Execution speed analysis is discussed. The problems of resource conflicts and branching are also addressed. Based on this discussion, the choice of an optimal clocking scheme is described.

Chapter 3 describes the algorithms which perform clocking scheme synthesis. An example clocking scheme for the HP-21MX computer is described, and compared to the manual design. Finally, an extension to the technique to handle general designs, including systolic arrays, is presented, and an example given.

The theory of pipeline synthesis is presented in Chapter 4. A data flow graph model for specifying input to the synthesis procedures is described. Then, the scheduling and resource allocation tasks with cost and performance constraints are discussed. Resource sharing across conditional branches is addressed, and an algorithm to color the data flow graph nodes to detect the possibility of resource sharing is presented.

Chapter 5 describes the pipeline synthesis algorithms based on the theory described in Chapter 4. Three types of polynomial-time algorithms are discussed, along with an exhaustive scheduling algorithm for optimal pipeline designs.

The insertion of delays into previously designed pipelines is the topic of Chapter 6. An optimal delay insertion algorithm is presented.

Finally, conclusions and future research are summarized in Chapter 7.

Chapter 2
Theory of Maximum Execution Overlap

2.1. Introduction

The main goal of this chapter is to develop a theory for the analysis and synthesis of a maximally-overlapped execution scheme for a digital system at the micro (register-transfer) level. By maximum overlap, we mean the following. *Each computation task uses a hardware resource no more than once* and *the data flow of a computation task moves in one direction* as fast as possible through a circuit. As soon as the current task moves to a later stage and frees a part of the circuit, the next task is initiated and executed on the free part of the circuit.

The emphasis of the theory to be developed in this chapter will be on *maximizing the performance while minimizing the change in the control and data flow of a given partial or complete design*. If a new set of operators were chosen for a potentially faster design, both the data and control flow must be altered to get the maximum advantage. Also the number and locations of the storage elements must be re-determined due to the different timing requirements of the new set of operators. Thus, the task would involve almost the same amount of work as the initial design task. For example, rewriting the microprogram or changing the value allocated to the operand and/or result registers automatically would involve changing the interconnections for both the data flow and control flow. In order to avoid such costly and time consuming iterations, in this chapter we consider adding or reconfiguring only storage elements, which can be done without altering the basic structure of the

original control and data flow and thus is considered to be transparent to the data flow and control flow analyzer. Assuming that the control sequence of the micro cycles is fixed, we consider two basic approaches to the problem:

1. For a set of chosen and allocated functional modules (for both the data path and the micro engine), add and connect the minimum number of storage elements necessary to achieve a certain level of performance.

2. For a completed design, add certain number of storage elements to the data path and/or micro engine in such a way that the performance increase will be maximized by virtue of maximum execution overlap of the micro cycles.

In any case, we try to maximize execution overlap of the micro cycles considering the time overhead due to branches, resource conflicts and data dependency relations. Maximum execution overlap can be achieved by synthesizing an optimal clocking scheme, which involves the following tasks:

- optimal assignment, relocation, addition or deletion of the stage latches,

- choice of an optimal clock period and the number and lengths of the clock phases (refer to Section 1.2.4), and

- optimal clock signal gating and routing.

In carrying out these tasks, we formulate the problem as a graph theoretic problem. Digital circuits are modeled by directed graphs which show the pathways of the data and control flow. By properly weighting the vertices and the directed edges, we can model the execution sequences of the micro cycles as tours on the graphs. Also, the time taken at each segment of the tours can be computed easily. Assigning and/or inserting overlap stage latches can be modeled as finding multiple edge cut sets. Once the locations of the stage latches are determined, then the optimal clock period and clock sequence can be computed considering the synchronization overhead discussed before.

2.2. Definition of the Clocking Scheme Synthesis Task for Maximum Execution Overlap

As we have discussed in Chapter 1, the general digital design problem is known to be computationally intractable. As one way of reducing complexity, synthesis of digital systems is usually partitioned into data path synthesis followed by control synthesis (this is true for both automated design systems [Parker 79, Estrin 78, Zimmermann 79] and human designers). In such design procedures, clocking scheme synthesis is constrained by both the data path design and the control design. Clocking scheme synthesis is carried out as one of the last tasks of control synthesis. For a given data path design and a control hardware design, the task of clocking scheme synthesis is as follows:

- choose an optimal clock period,
- determine an optimal number and length of the clock phases, and
- assign clocked control signals to clock phases and route to the data path.

However, most of the important parameters determining the execution speed of digital systems are fixed during the data path and control design. Thus, optimality of the design can be guaranteed only if clocking scheme synthesis is done concurrently with both the data path design and other parts of the control design. For example, cheaper data path designs often require more elaborate clocking schemes and therefore a final solution to the data paths cannot be chosen until the clocking cost is examined (and indeed, until the entire cost including control is examined).

In this thesis, we shift the occurrence of clocking scheme synthesis to somewhat earlier phases of the design procedure in order to synthesize near-optimal digital systems. We define the task of clocking scheme synthesis as follows:

The inputs are

(i) a partial data path and control design with chosen functional modules and minimum required storage elements,[7]

(ii) types of micro cycles (e.g., microinstruction formats or Node-Module-Range bindings [Knapp 83], which specify the direction and propagation time of data values through functional modules during micro cycles), and

(iii) expected sequences of micro cycles to be executed.

The constraints are

(i) minimum execution speed of the micro engine,

(ii) maximum number (or total bit width) of storage elements, and

(iii) maximum number of clock phases.

The outputs produced are

(i) sssignment, insertion or deletion and interconnection of storage elements necessary to obtain a certain execution speed (or speed to cost ratio),

(ii) minimum and optimal (not necessarily distinct) clock periods to maximize the execution speed,

(iii) the optimal number and length of clock phases, and

(iv) clock signal routing to stage latches.

---

[7]For any data path, the minimum number of storage elements is determined by the maximum number of live values [Aho 77] at any time. In most cases of computer CPU designs, the registers (e.g., ACC, MAR, and I/O buffer) and the main memory which the machine language programmer can directly access are the minimum set of storage elements. For control hardware, it can be either the μ-PC or the microinstruction register.

We consider three cases of partial data path designs. The first case includes designs which need more storage elements to be allocated and connected in order to satisfy the machine behavior. The second case includes designs which have already been completed and only the connections from and to the storage elements are partly or completely undone for the purpose of reconfiguration of the interconnections. The third type includes completed designs. Even for completed designs, we may need to add or delete storage modules in order to increase performance at minimum cost or to decrease cost at minimum sacrifice of speed.

In any case, the objective of the clocking scheme synthesis task is, while satisfying all the design constraints and desired goals, to maximize the execution speed of the system by optimally configuring, adding and/or deleting storage modules and consequently determining an optimal clock period and number of clock phases.

There is no absolute order in carrying out these tasks. The result of each task may affect the results of one or more of the others. For example, choice of an optimal clock period and determination of the optimal number of clock phases depend on the result of optimal stage partitioning. Also, the maximum allowed number of clock phases (due to clock generator cost and/or clock signal routing complexity) will affect both the choice of an optimal clock period and optimal stage partitioning. For this reason, *a unified technique is strongly desired in order to examine the attributes of all the design decisions in parallel.*

2.3. The Problem Formulation

In this section, we first discuss the problem definition and modeling of the clocking scheme synthesis task for maximum execution overlap. Next we extract the parameters affecting the performance of the micro-level maximum-execution-overlap scheme.

2.3.1. Specifying the functioning times of digital circuits

Timing behavior of a circuit can be represented as the sequence of activations of hardware modules in the circuit and the timing behavior of the individual modules. For example, the execution time of a computation through a circuit can be analyzed by tracing and adding up the delay time of the data and control flow through the modules which are activated during the computation. In order to analyze the timing behavior of a circuit, we first need to identify modules as the basic units of a hardware circuit which have well-defined and independent timing behavior.

Definition 2.3.1: A **logical module** is a set of physical hardware elements (e.g. resistors, transistors, gates, etc.) which can perform a certain complete function (either operations or storage) without any resource contention with other functions at any time.

An adder chip or a pass gate can be considered to be a logical module. A bidirectional bus must be considered to be a separate logical module since it cannot be considered to be a part of any one module connected to it. A set of interconnection lines which are always used together to transfer a certain value can also be considered to be a logical module.

Logical modules may be either physically separated or share common physical hardware modules. A physical hardware module which can perform more than one function at the same time can be considered to be multiple

logical modules. *A register can be considered as two logical modules, read and write modules, if it can be read and written simultaneously without any conflicts in using its control and data lines.* However, a memory chip cannot be read and written at the same time,[8] and thus is considered to be a single logical module.

In this sense, a logical module can be considered a unit hardware resource whose timing and functional behavior can be independently, unambiguously defined. The delay time of a module is defined as follows, considering the race and clock skew problems we have discussed in Section 1.4.2.

Definition 2.3.2: The **module propagation delay** of a module is the maximum propagation delay for all possible pairs of an input and an output port (i.e. critical path delay) of the module and for all possible values of input data.

The micro cycle time or minor cycle time, which we have discussed in Section 1.2.5, can be computed by summing up the propagation delays of the control and data flow through the modules along the execution paths. Also, resource contention between executions of the micro cycles can also be represented in terms of the logical modules. From now on, the term module will always imply logical module, unless otherwise specified.

Using the concept of *logical modules*, we can model a digital circuit as a weighted, directed graph (circuit graph), where the vertices of the graph represent modules and the directed edges represent all the possible pathways for both the control and data values between the modules in the circuit. The purpose of the circuit graph is to connect the control and data path hardware together.

---

[8] Two-port memory chips have separate read and write address and data lines and are considered to be two logical modules.

Definition 2.3.3: A **circuit graph**, G = (V,E), is a directed graph where the set of vertices, V, represents modules, and the set of directed edges, E, represents the pathways for data and control values between modules. A directed edge, e(i,j), belongs to E if any output port of module i is connected to any input port of module j. The vertices are weighted with the propagation delays of the modules and the edges are weighted with the bitwidths of the interconnections.

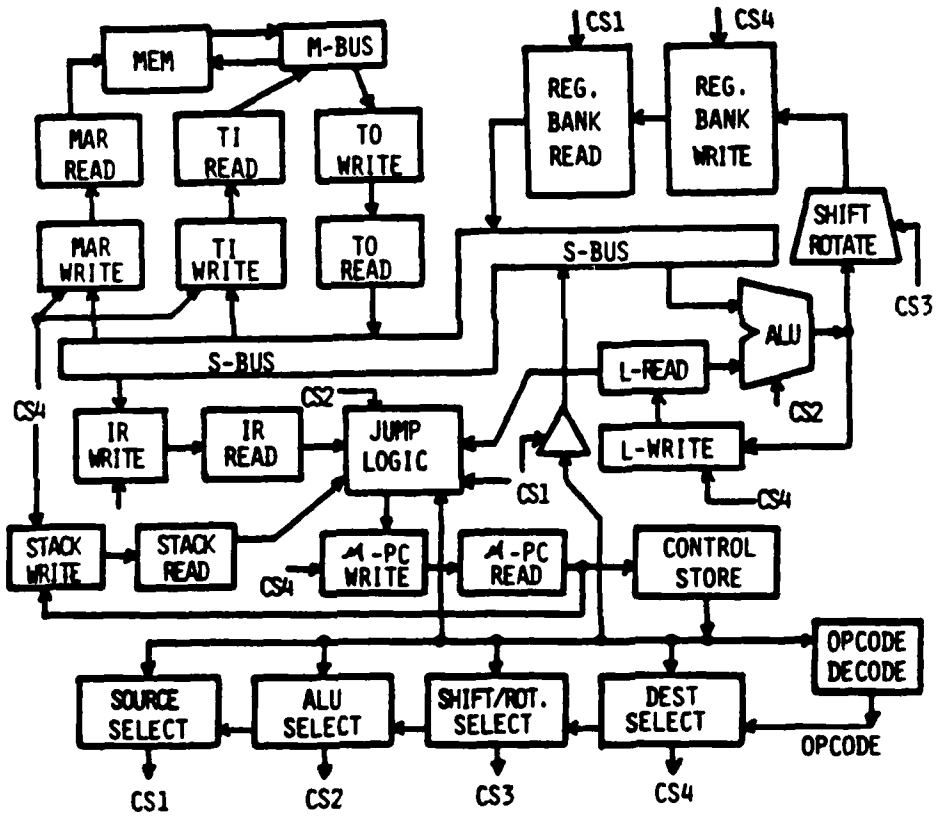


Figure 2.3-1: A circuit graph of a microprogrammed CPU (HP-21MX).

By weighting the vertices with the propagation delays of the modules, the upper bound of the propagation delays of data values and control signals along

any path in the circuit can be computed. The vertices for interconnection lines with non-negligible propagation delays must be added. The bitwidths of the data and control flow can be computed easily using the edge weights. In the case of a partially designed system, the necessary interconnections for the flow of control and data values specified in the data flow graph and control graph might be missing. Figure 2.3-1 shows an example of a circuit graph without the vertex and edge weights.

2.3.2. Modeling sequencing behavior of micro cycles

For the analysis of the sequencing and timing behavior of the micro cycles, we introduce two directed graphs, the **micro-cycle graph** and the **chain of minor cycles**. They are based on the circuit graph and model the pattern of resource usage and timing of the micro cycles.

2.3.2.1. The micro-cycle graph

In order to model the pattern of resource usage and the execution time of the micro cycles, we construct edge-weighted, vertex-weighted, acyclic digraphs, called micro-cycle graphs. The micro-cycle graphs are subgraphs of the circuit graph.

Definition 2.3.4: For a given circuit graph, $\mathcal{G}(\mathcal{V},\mathcal{E})$, the micro-cycle graph for a set of one or more micro cycles, G(V,E), is a **rooted** subgraph of $\mathcal{G}$ with no directed cycles. The set of vertices, $V \subseteq \mathcal{V}$, and the set of directed edges, $E \subseteq \mathcal{E}$, represent only the modules and interconnections **activated by** and **necessary to** the execution of the micro cycles in the set.[9] The vertices and edges are weighted in the same way as in the circuit graphs.

---

[9]Read modules whose outputs are always enabled and write modules which are not written during the execution of the micro tasks in the set are not included, although the values contained in them may be needed by the micro cycles.

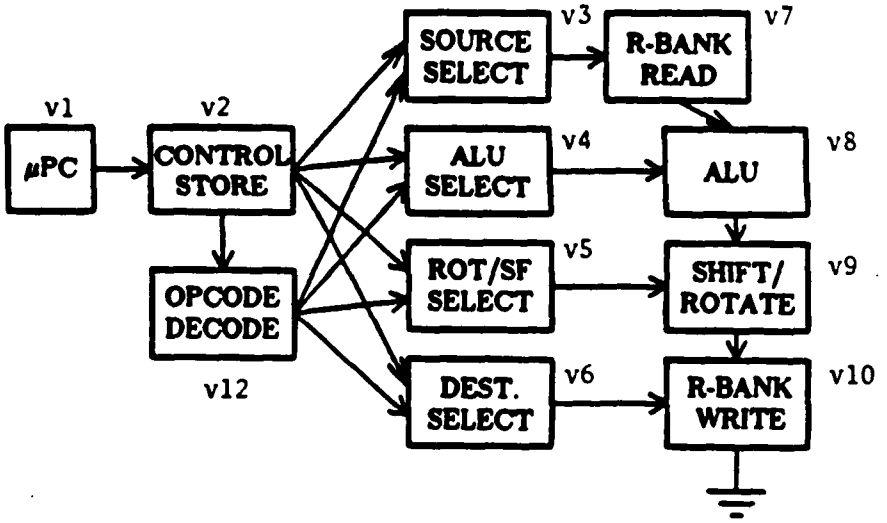
The micro-cycle graphs are rooted by common root(s), which represent the common starting point of every micro cycle. In general, for any synchronous sequential circuit or finite state machine [Friedman 75], there must be memory and/or delay elements in order to prevent state-change races and/or to control the time intervals between state changes. Among the memory or delay elements, we choose a subset of them as the starting point of every cycle. For a microprogrammed micro engine, the starting point can be either the μ-PC or the microinstruction register. For a hardwired sequencer, the starting point can be either the state counter or feedback memory.

Any branch micro cycle (e.g. JMP or SKIP) must be separated from non-branch micro cycles since it changes the control flow after it is completed and delays initiation of the next micro cycle until its completion.

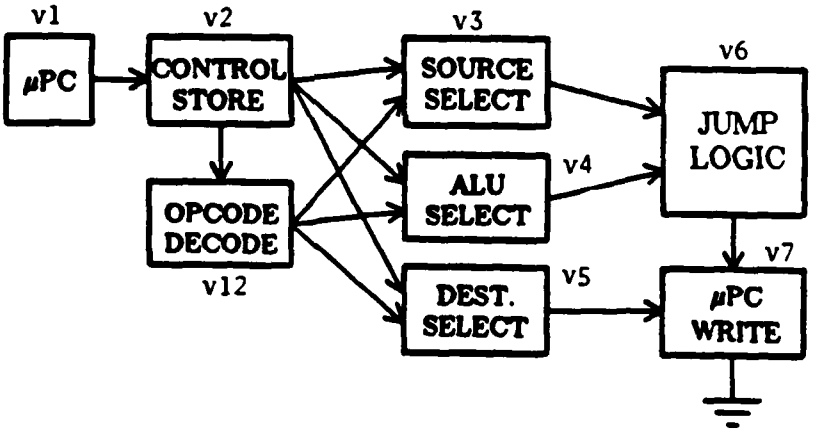
In the case of complex digital systems, there can be more than one type of micro cycle. For example, for a microprogrammed CPU, there can be as many micro-cycle graphs as the number of microinstruction formats. However, more than one type of micro cycles can be represented by a single micro-cycle graph as long as the resulting micro-cycle graph has no internal cycles.

Figure 2.3-2 shows two micro-cycle graphs derived from the circuit graph of Figure 2.3-1. Both are rooted at the μ-PC. Figure 2.3-2-(a) is the micro-cycle graph for non-branch type microinstructions. The execution sequence of this type of microinstruction is

1. increment the PC (v1),
2. fetch the micro cycle pointed to by the PC (v2),
3. decode the control fields of the fetched micro cycles: opcode (v12), operand register address (v3), ALU function code (v4), rotate/shift function code (v5), and result register address (v6),
4. fetch operand from the selected register (v7),



(a) Micro-cycle graph for non-branch microcycles.



(b) Micro-cycle graph for branch microcycles.

Figure 2.3-2: Examples of the micro-cycle graphs in the circuit graph of Fig. 2.3-1.

5. perform the selected ALU operation (v8),

6. perform the selected rotate/shift operation (v9), and

7. store the result in the selected register (v10).

An example of the execution sequence of a branch micro cycle corresponding to the micro-cycle graph of Figure 2.3-2-(b) is as follows:

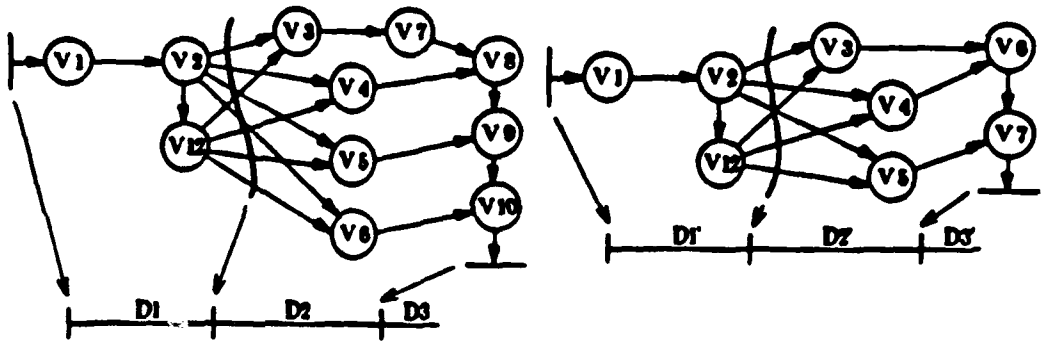
1. increment the PC (v1),
2. fetch the branch micro cycle pointed to by the PC (v2),
3. decode the control fields of the fetched micro cycle: opcode (v12), condition select (v3), branch address modification (v4), and branch type (v5),
4. select a test condition and select the full jump address (v6), and
5. load the PC with the jump address and, at the same time, if it is a CALL to a subroutine, save the current PC contents in the stack (v7).

The micro-cycle graph shows the sequence of activation of the modules during a micro cycle. The micro-cycle graphs can be used to determine the locations for the stage latches which either have to be added or already exist. The locations and connections of the stage latches determine the interstage propagation delays between the stage latches. Also, by weighting the edges with the bitwidths of the corresponding interconnection lines, the bitwidths of the added stage latches can be computed.

2.3.2.2. The chain of minor cycles

Once the locations and connections for the stage latches are determined, the interstage propagation delays are also determined and thus the minimum requirements of clocking and timing for the micro cycles are determined. These basic timing requirements are modeled by one or more line graphs - more precisely, chains (chains of minor cycles) - which show the minimum required execution time of minor cycles as well as the minimum required clock period.

Figure 2.3-3 shows examples of the chains of minor cycles derived from the results of stage partitioning of the micro-cycle graphs of Figure 2.3-2. The



Di's are the stage propagation delays.

(a)Non-branch micro cycles (b)Branch micro cycles

Figure 2.3-3: The chains of minor cycles derived from the results of stage partitioning.

locations of the stage latches are indicated by the edge-cut lines in the micro-cycle graphs. In case (a), the stage latches are the μ-PC, the latches next to the control store (v2) and the opcode decoder (v12), and the register bank (v10) (refer to Figure 2.3-2).

The chain of minor cycles shown in Figure 2.3-3-(a) shows that each non-branch micro cycle consists of three minor cycles and needs three clock ticks to complete. The minimum time intervals between clock ticks, D1 and D2, are determined by the critical path delays between the stage latches. For example, D1 is equal to the delay time through the modules, v1, v2, and v12, plus the delay time of the stage latches. We will discuss the detailed clocking and timing requirements of minor cycles in Section 2.3.3.

At the end of the chain, D3 must be added in order to consider the completion time of all the effects of an execution of a micro cycle, although it is not explicitly specified in the micro-cycle graph. It is necessary to analyze

the effect (data dependency, resource contention, etc.) of a micro cycle on its successor. For example, if the next micro cycle reads the result of the current micro cycle which will be stored in the third stage latch (e.g., v_{10} of micro-cycle graph (a) or v_7 of micro-cycle graph (b) in Figure 2.3-2), then the next micro cycle can only read the correct value after the buffer has been clocked and the stored values propagated to its outputs.

The chain of minor cycles can be used to determine a major clock period and the number and lengths (phase lag) of the clock phases which clock the stage latches and execute the minor cycles. Resource conflicts between the minor cycles can also be represented by attaching the module names used by the minor cycles to corresponding edges of the chain.

2.3.3. The minimum possible length of a clock phase

In any design, minimizing the time interval between the clocking of any two storage elements is desired in order to increase the speed of the circuit. However, the time interval between clock pulses must be long enough so the logic in between any two storage elements can read inputs, complete execution, and latch results.

The execution time of the logic in between any two latches can be computed by computing the length of the critical path between the latches. We define the necessary clock timing for each case of three most widely used types of latches.

Let us first define several timing variables for the clarity of discussion.

T_{min} The critical path delay of the logic between a pair of stage latches. This is the minimum required amount of time for a stage of combinational logic to complete processing input and produce all the outputs.

T_{stage} — The actual amount of time available for a stage to complete processing and produce all the outputs. T_{stage} must be longer than T_{min}.

W — The width of the clock pulse.

Dss — Storage set-up time for edge-triggered type latches [Hafer 83]. Inputs to edge-triggered latches must be available earlier than the clock tick by Dss.

Dsp — Storage propagion time. For level-sensitive latches, the clock pulse width must be longer than Dsp and the input must remain stable until the clock becomes inactive in order to latch the input safely. For edge-triggered latches, the new output is available Dsp units of time after the clock tick.

T_{cy} — The time interval between clocking the input latch and the output latch of a stage.

Figure 2.3-4 illustrates clock timing when level-sensitive latches with active-high clock inputs are used. The latches start propagating inputs to the outputs whenever the clock goes high. If and only if the clock pulse is longer than the propagation delay of the latches, the latches hold values on their outputs the same as the inputs when the clock goes low. Therefore, the clock pulse width, W, must be longer than the latch propagation delay, Dsp, and T_{cy} must be longer than T_{min} + Dsp. Thus,

$$T_{cy} \geq T_{min} + Dsp \qquad (2.3.1)$$

Figure 2.3-5 shows clock timing with edge-triggered latches. For edge-triggered latches, inputs must be available Dss before clock tick, and new outputs are available Dsp after the clock tick. Therefore,

$$T_{cy} \geq T_{min} + Dss + Dsp \qquad (2.3.2)$$

Figure 2.3-6 illustrates clock timing with master-slave latches. Master-

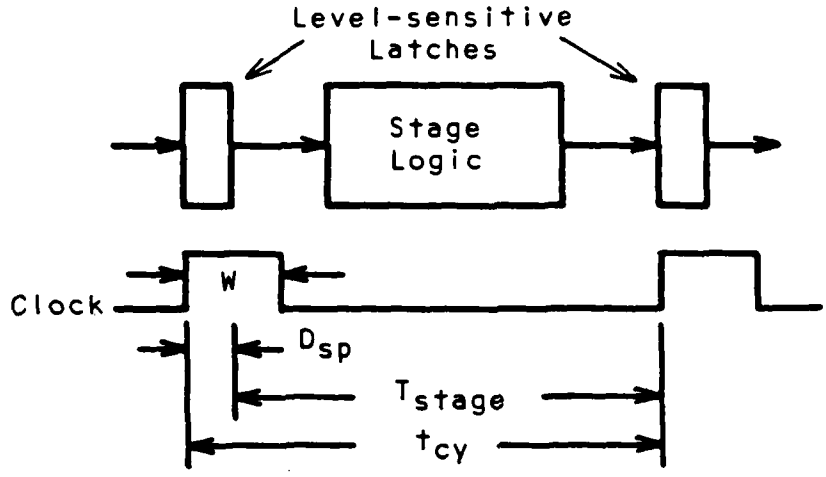


Figure 2.3-4: Clock timing for level-sensitive latches.

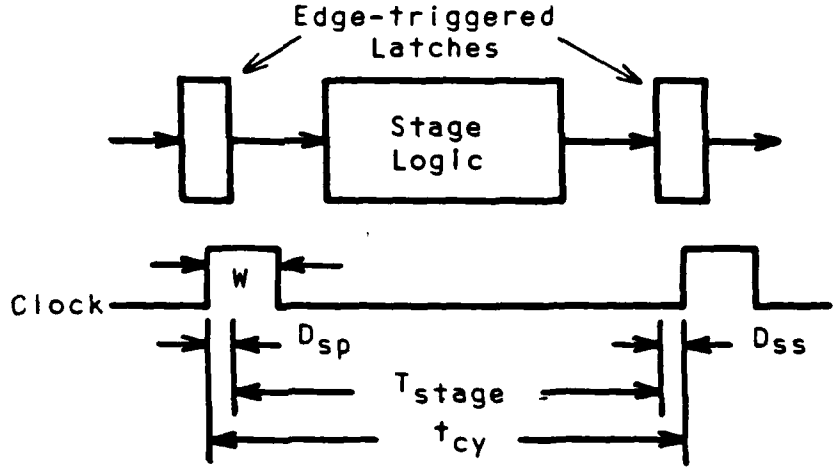


Figure 2.3-5: Clock timing for edge-triggered latches.

slave latches accept input into the master stage while the clock is active and start propagating the input to the output when clock goes inactive. Let Dsp(slave) be the propagation delay through the slave stage. Then, assuming that the clock pulse width is longer than the propagation delay of the master stage,

$$T_{cy} \geq Tmax + W + Dsp(slave) \qquad (2.3.3)$$

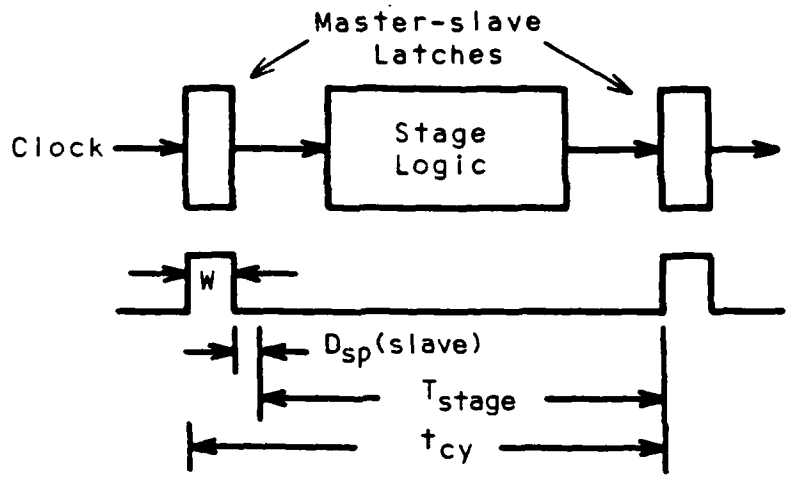


Figure 2.3-6: Clock timing for master-slave latches.

2.3.4. A static clocking assumption

In the worst case, we may have as many distinct chains of minor cycles as the number of micro-cycle graphs. For an optimal design, this situation may require as many distinct clocking sequences with different clock periods and different numbers and lengths of clock phases. Especially in the cases where execution overlap is extensively used, all the different clocking sequences may have to be overlapped and thus as many separate clock generators are required. A complex initiation and termination control mechanism for the clock sequences is required in order to prevent conflicts in the usage of both the hardware resources and the data values between micro cycles using different clocking sequences.

In actual designs, this is not realistic and seldom happens because of the cost and control complexity of the clock generator(s) and clock signal gating and routing. For this reason, a static clocking scheme - a fixed clocking

sequence with a fixed number and lengths of clock phases - is usually used in actual designs. In this thesis, we will focus on synthesizing an optimal static clocking scheme for all micro-cycle graphs.

2.3.5. Sequencing behavior of overlapped micro cycles

In this section, we analyze the sequencing behavior of the overlapped micro cycles according to their timing, pattern of resource usage, and interactions (data dependencies and resource conflicts) between them.

The **maximum initiation rate of micro cycles** is defined as the maximum possible number of initiations of micro cycles during some unit time period when there are neither branch micro cycles nor resource/data conflicts between micro cycles.

Figure 2.3-7 shows examples of micro cycle sequencing with different sequences of clock phases. We assume a static clocking sequence. (a) does not use any overlap, hence there is only one stage latch and one stage. The micro cycle times of (b) through (d) are longer than that of (a) due to the propagation delays of the stage latches. In any case, the maximum initiation rate of the micro cycles is the same as the clock rate, t_{cy}. The clock period must be longer than the longest interstage propagation delay in order to ensure that no two micro cycles occupy the same stage at the same time. Figure 2.3-7-(b) uses the shortest clock period possible, which is 2.

A **branch** micro cycle changes the normal sequential execution sequence and thus delays the fetch of the next micro cycle until the earliest fetch clock cycle after the completion of the branch. Due to this branch time overhead the shortest clock period does not guarantee the fastest overall initiation rate.

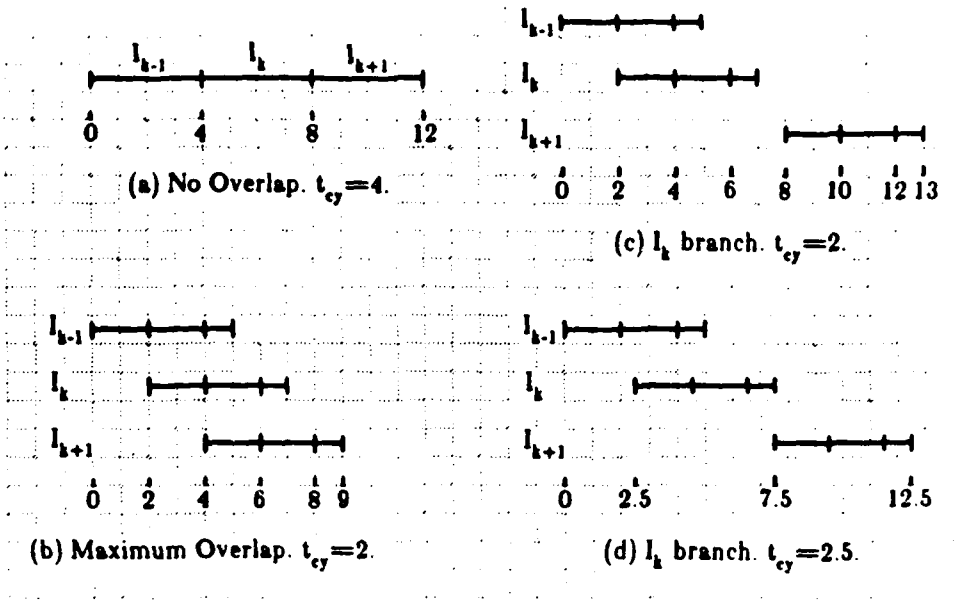


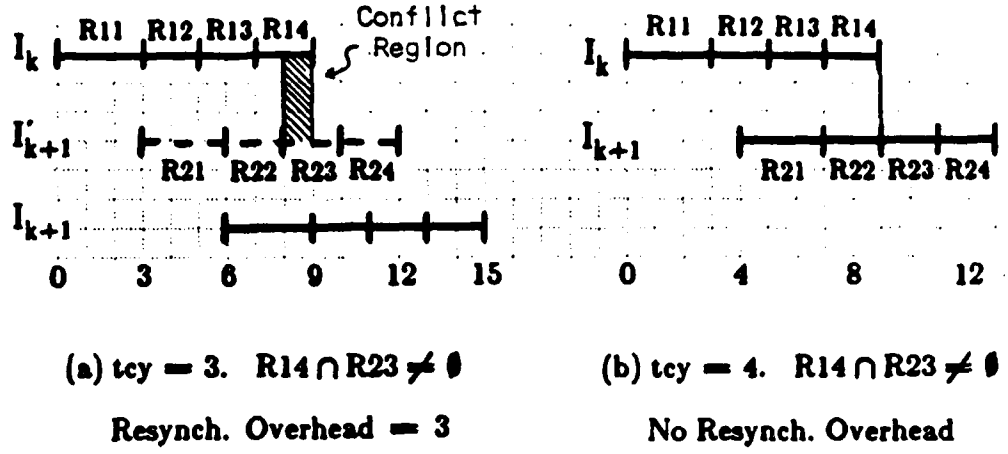
Figure 2.3-7: Examples of micro cycle sequencing and clocking.

Definition 2.3.5: Any event that delays initiation of a new micro cycle is called **resynchronization**. The time delay or overhead due to a resynchronization is called **resynchronization overhead**.

Even increasing the clock period may result in a faster execution if it can reduce the resynchronization overhead. As shown in Figure 2.3-7, the total execution time of (d) is shorter than that of (c) in executing I_{k-1} through I_{k+1} since, in (c), the instruction can only be initiated every 2 time units, delaying I_{k+1}. Thus, the overall initiation rate will also depend on the frequency of the branch micro cycles. Therefore, determination of the optimal clock period should consider the resynchronization overhead due to branches.

Resource and data contention between micro cycles are other causes for delaying initiation of new micro cycles. If there is either data or resource contention between any two micro cycles, fetching the later micro cycle must be delayed until its initiation does not cause any contention with its predecessor. The delay time is dependent on both the clock period and the pattern of data and/or resource contention between the micro cycles. These cases are shown in Figure 2.3-8. In case (a), if I_{k+1} is initiated as the dotted cycle (I'_{k+1}), then there will be a resource conflict or data dependency violation between the minor cycles using resources R_{14} and R_{23}. Case (b) does not have any resynchronization overhead. This shows that the resynchronization overhead can be reduced by choosing a proper clock period.

The time required for resynchronization may depends on **the length of the clock phases** even if the same clock period is used. Increasing the intervals between the clock phases may reduce the number of distinct clock phases without increasing the clock period. Figure 2.3-9-(b) shows a clocking sequence which is exactly the same as the chain of minor cycles, which requires four distinct clock phases. Figure 2.3-9-(c) has only two distinct clock phases



(a) tcy = 3. $R14 \cap R23 \neq \emptyset$
Resynch. Overhead = 3

(b) tcy = 4. $R14 \cap R23 \neq \emptyset$
No Resynch. Overhead

Figure 2.3-8: Resynchronization overhead due to data/resource contention.

with the same initiation rate as (b) regardless of the resynchronization overhead. However, in (d), the branching overhead is longer than that of (b) or (c) by one clock period (4 time units) and thus the overall initiation rate is lower. Although there might be some difficulties in gating and routing a single phase clock to multiple stage latches selectively, reducing the number of clock phases may reduce the physical routing problem significantly.

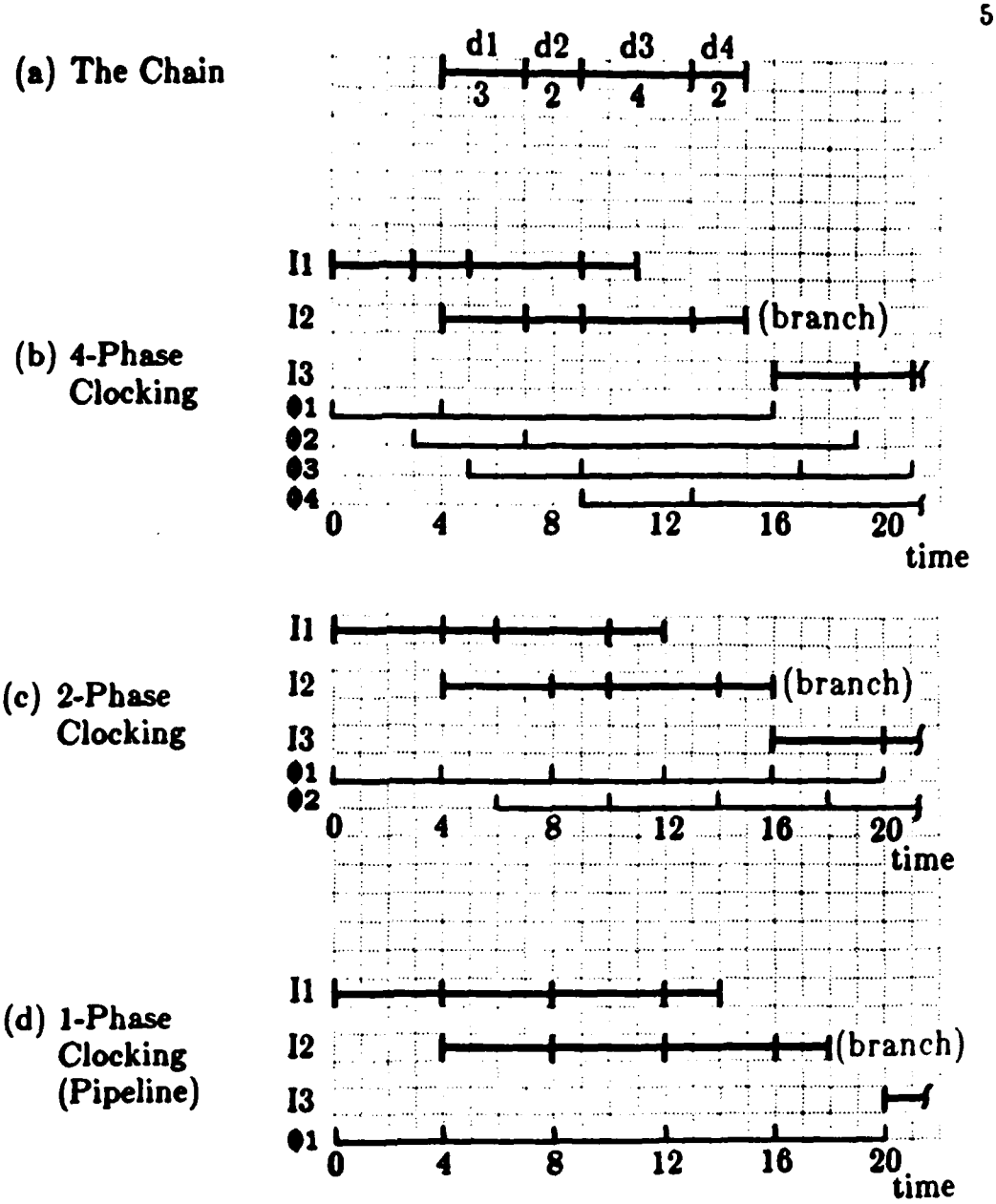


Figure 2.3-9: The number of clock phases vs. resynchronization overhead.

2.4. Synthesis of Clocking Schemes for Maximum Execution Overlap

This section contains discussion and theoritical results of clocking scheme synthesis for maximum execution overlap.

We analyze optimal stage assignment and choice of an optimal clock period. Also, determination of an optimal number of clock phases and their length is also analyzed. These analyses are carried out under two different goals: (i) to find an absolute optimal solution and (ii) to find an optimal solution with respect to certain constraints. Simple and efficient algorithms to determine optimal positions of the stage latches and optimal number of clock phases are developed in Chapter 3. We believe that we can easily extend these results to analyze data path cycle with loops and, furthermore, to analyze more general cases of system timing styles.

2.4.1. Definition of variables[10]

L_i	The i-th stage latch of the micro-cycle graph, which is clocked by the i-th clock phase.
C_i	The control/data path stage in between L_i and L_{i+1}.
d_i	The minimum possible stage time for the i-th stage, C_i (The minimum possible T_{stage} defined in Section 2.3.3).
d_{max}	$\max\{d_1, d_2, \ldots, d_m\}$ where m is the number of stages.[11]
d_S	$d_S = d_1 + d_2 + \ldots + d_m$

[10] The reader is urged to skip this section and refer back to it while reading this chapter.

[11] m is the number of stages of the micro-cycle graph with the largest number of stages. We call such a system an **m-stage system**.

I_i	A micro cycle as an instance of an execution of a micro task (e.g., an execution of a microinstruction).
ϕ_i	Clock phase i. Used to latch L_i.
$\phi_j(k)$	Time when clock phase j latches L_j to execute a micro cycle I_k.
D_i	The actual stage time assigned to the i-th stage.
	$D_i = \phi_{i+1}(j) - \phi_i(j) \geq d_i, \quad 1 \leq i < m.$
D_{max}	$\max\{D_1, D_2, \ldots, D_m\}$
D_S	$D_1 + D_2 + \ldots + D_m$
t_{cy}	Clock period. $t_{cy} \geq D_{max}$

2.4.2. Execution speed analysis

Determination of the Minimum Clock Period

The minimum clock period for a multiple stage system is determined by the interstage propagation delays. In order to ensure correct sequencing of micro cycles, the minimum clock period should be longer than the longest interstage propagation delay [Cotten 65, Ramamoorthy 77, Chen 75].

Lemma 2.4.1: For an m-stage system, M, with the minimum possible stage times $(d_1, d_2, \ldots, d_m)$, the minimum possible clock period must be longer than $\max\{d_i\}$, or $\min(t_{cy}) = d_{max}$. (Refer to Figures 2.3-7, 2.3-8, and 2.3-9.)

Proof : The clock period must be long enough so that (i) each stage can have enough time to execute a given subtask and (ii) there is no collision between micro cycles at any stage at any time.

From the definition of $\phi_i(j)$, $\phi_i(j+1) = \phi_i(j) + t_{cy}$, $\forall$ i,j, $1 \leq i \leq m$

Thus, $\phi_i(j+1) = \phi_i(j) + t_{cy} \geq \phi_i(j) + d_i$, $\forall$ i,j, $1 \leq i < m$

Therefore, $t_{cy} \geq d_i$, $\forall$ i, $1 \leq i < m$ (1)

Also, $\phi_m(j) + t_{cy} \geq \phi_m(j) + d_m$.

Therefore, $t_{cy} \geq d_m$ (2)

From (1) and (2), $t_{cy} \geq d_i$, $\forall$ i, $1 \leq i \leq m$. Therefore $\min(t_{cy}) = d_{max}$.

□

Execution Time of a Sequence of Micro Cycles

For a sequence of micro cycles, the execution time is defined as the *time from the fetch clock phase for the first micro cycle in the sequence to the earliest fetch clock phase after the completion of the last micro cycle in the sequence.* For an execution sequence of n micro cycles, if there are no branch micro cycles and there is no other resynchronization overhead, then the execution time, T, is computed as the sum of the following:

$A = (n-1)\cdot t_{cy}$ for the first (n-1) micro cycles which are initiated every t_{cy} period.

$B = \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy}$, which is the execution time of the last micro cycle.

$$T = A + B = (n - 1 + \left\lceil \frac{D_S}{t_{cy}} \right\rceil) \cdot t_{cy} \qquad (2.4.1)$$

Delay Due To Branching

Any branch cycle delays the fetch of the next micro cycle until the first fetch micro cycle after its completion. The difference between the fetch time of the next micro cycle *after a branch cycle* and *after a non-branch cycle* is defined as **branching overhead**.

Lemma 2.4.2: Let M be an m-stage machine with a multiphase clocking scheme (D_1, D_2, ... , D_m) and a clock period t_{cy}. For two execution sequences, S_1 and S_2, let S_2 be the same as S_1 with some non-branch cycle, I_j, $1 \leq j < n$, replaced with a branch cycle, I_j'. The execution time of S_2 is longer than that of S_1 by

$$t_{cy} \cdot \{ \left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 \}$$

Proof: (Refer to the figure below.) The only execution interval affected by the replacement is between $\phi_1(j)$ and $\phi_1(j+1)$, i.e., $\phi_1(j+1)$ is to be delayed. Let T1 and T2 be $\phi_1(j+1)$'s before and after the replacement, respectively. Then,

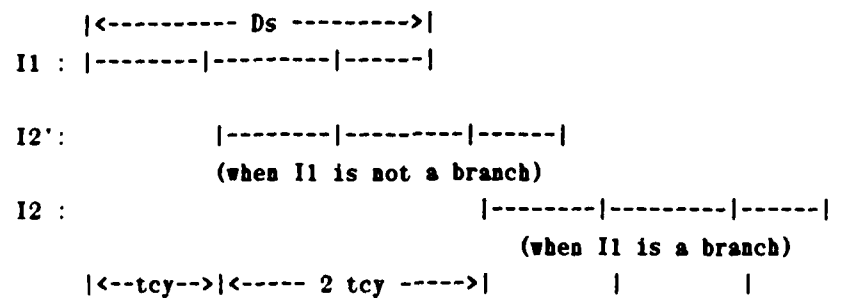
$$T1 = \phi_1(j) + t_{cy} \tag{1}$$

Since I_j' is a branch, $T2 \geq \phi_m(j) + D_m = \phi_1(j) + D_S$ (2)

From (2), we get: $T2 = \phi_1(j) + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy}$

Therefore, (branching overhead) $= T2 - T1 = \{ \left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 \} \cdot t_{cy}$.

□



If there are n_b branch micro cycles, then the total branching overhead is

$$n_b \cdot \{ \left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 \} \cdot t_{cy} \qquad (2.4.2)$$

Thus, the difference in execution times for two sequences of micro cycles is a function of the cycle time and the total interstage times. In actual systems, there may be several types of branch micro cycles with different execution times (no more than 4 types in most cases of micro-sequencers). Typical types of branch micro cycles which may have different execution times are conditional branch, unconditional branch, decode branch and *sense and skip*. In such cases, we can compute the branching overhead for each type of branch micro cycle. For example, let D_{Si} be the execution time span of type-i branch micro cycles over the sequencing chain. Then the branching overhead of type-i branch micro cycles is $\{ \left\lceil \frac{D_{Si}}{t_{cy}} \right\rceil - 1 \} \cdot t_{cy}$.

In fact, any other type of resynchronization overhead can be computed by replacing D_S with the corresponding time overhead. For this reason, we will focus on analyzing the effect of branches on the execution time of micro cycles. All the following analyses can be directly applied to execution sequences with any type of resynchronization.

Execution Time of an Execution Sequence with Branches

The execution time of an execution sequence of n micro cycles with n_b

branch micro cycles can be calculated as the sum of the execution time of n non-branch executions and the branching overhead for n_b branches.

Theorem 2.4.3: On an m-stage system M with a multiphase clocking scheme (D_1, D_2, ... , D_m) and clock period t_{cy}, the total execution time of a sequence of n micro tasks where n_b of them require resynchronization is

$$\tau = \{(n-n_b-1) + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot (n_b + 1) \} \cdot t_{cy}$$

Proof :

1. Every non-branch micro cycle except the last one is fetched and executed at the clock rate, t_{cy}.
2. The last micro cycle execution takes $\left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot t_{cy}$
3. By 1 and 2, an execution sequence of length n with all non-branch executions is executed in time $\{\left\lceil \frac{D_S}{t_{cy}} \right\rceil + (n-1) \} \cdot t_{cy}$ (1)
4. By Lemma 2.4.2, the time overhead caused by replacing n_b non-branch executions with branch executions is $n_b \cdot \{\left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1\} \cdot t_{cy}$ (2)
5. Replacing the last micro cycle, I_n, with a branch micro cycle does not change the execution time, since there is no overlapped execution afterwards anyway.

Therefore, the execution time = (1) + (2), or

$$\begin{aligned} \tau &= \{\left\lceil \frac{D_S}{t_{cy}} \right\rceil + (n-1)\} \cdot t_{cy} + n_b \cdot \{\left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1\} \cdot t_{cy} \\ &= \{(n-n_b-1) + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot (n_b + 1) \} \cdot t_{cy} \end{aligned}$$

□

Theorem 2.4.3 shows the relationship between the execution speed, the number of branches, the clock period and the length of the clock phases.

Modification for Micro Tasks with Different Execution Times

As mentioned before, the system may have several types of branch micro cycles with different execution time spans. Suppose that there are j different types of branch micro cycles. Let n_{bi}, $1 \leq i \leq j$, be the number of type i branch micro cycles with execution time span D_{Si} out of n_b total branch micro cycles. Then, we can replace Equation (2.4.2) with

$$\sum_{i=1}^{j} n_{bi} \cdot \{ \left\lceil \frac{E_i}{t_{cy}} \right\rceil - 1 \} \cdot t_{cy} \tag{2.4.3}$$

Also non-branch micro cycles may have different execution time spans. Assuming that we know the execution time span of each type of micro cycle and the execution sequence of micro cycles, we can also generalize Equation (2.4.1). In order to generalize Equation (2.4.1), we only need to consider cases where the execution of I_l, for some l, $l < n$, completes later than I_n. Since any branch micro cycle, I_i, $1 \leq i < n$, must complete execution before I_{i+1} starts, we can exclude branch micro cycles from this special case computation. Therefore we only need to consider such I_l's that there is no branch micro cycle in between I_{l-1} and I_n. Then we can replace Equation (2.4.1) with

$$(n-1) \cdot t_{cy} + \max_{l} \{ \left\lceil \frac{E_l}{t_{cy}} \right\rceil - (n-l) \} \cdot t_{cy} \tag{2.4.4}$$

where $1 \leq l \leq n$, and there is no branch micro cycle in between I_{l-1} and I_n.

Using Equations (2.4.3) and (2.4.4), we can fully generalize all the previous analyses to dynamic clocking analysis, where the micro cycles may have different execution time spans. However, as we can see by Equations (2.4.3) and (2.4.4), dynamic clocking analysis can simply be considered to be a general case of static clocking analysis. Exactly the same approach and methods can be used for both analyses by simply adjusting several variables

and/or constants as is done in Equations (2.4.3) and (2.4.4). For this reason, we will focus on static clocking analysis.

2.4.3. Maximum execution speed analysis

Execution speed of a multi-stage system is a function of:

1. the stage propagation delay d_i's,
2. the clock period t_{cy},
3. the clocking scheme (D_i's), and
4. the given execution sequence, S.

In this section, we analyze the effects of each of these execution speed parameters.

Determination of an Optimal Clock Period

The following lemma shows that a clock cycle which is an integer dividend of the execution span, D_s, results in a local optimal execution speed.

Lemma 2.4.4: Let $\mathcal{M}$ be an m-stage digital system with a clocking scheme (D_1, D_2, ... , D_m). On $\mathcal{M}$, Let $\mathcal{T}_t(x)$ be the execution time of a sequence of n micro cycles with a clock period x. For any execution sequence,

$$\mathcal{T}_t(\frac{D_S}{k}) \leq \mathcal{T}_t(\frac{D_S}{k'}) \text{ for any integer } k, \left\lceil \frac{D_S}{D_{max}} \right\rceil > k \geq 1, \text{ and real } k', 0 < k' \leq k.$$

Proof: Let n_b be the number of branch micro cycles out of a total of n micro cycles to be executed. By Theorem 2.4.3,

$$\mathcal{T}_t(\frac{D_S}{k'}) - \mathcal{T}_t(\frac{D_S}{k}) = (n-n_b-1)\cdot(\frac{D_S}{k'} - \frac{D_S}{k}) + (n_b + 1)\cdot\{\lceil k' \rceil \cdot \frac{D_S}{k'} - D_S\} \quad (1)$$

By evaluating the range of each component in (1), we get:

1. $n_b \leq (n-1)$ and $n - 1 - n_b \geq 0$

2. $n_b \geq 0$, $n_b + 1 \geq 1$

3. $0 < k' \leq k$, $\frac{D_S}{k'} - \frac{D_S}{k} \geq 0$

4. $\lceil k' \rceil / k' \geq 1$, $\lceil k' \rceil \cdot \frac{D_S}{k'} - D_S \geq 0$

By Equation (1) and from 1 to 4, $\tau_t(\frac{D_S}{k'}) - \tau_t(\frac{D_S}{k}) \geq 0$.

Therefore, $\tau_t(\frac{D_S}{k}) \leq \tau_t(\frac{D_S}{k'})$.

□

Lemma 2.4.4 shows that *a clock cycle which is an integer dividend of the execution span of a micro cycle, D_S, always results in equal or shorter execution time than any other clock cycle that is longer.* Now we shall show that *execution time is a discontinuous function of the clock cycle if there is any resynchronization overhead.* We show this by examining the execution time with branches with a clock cycle which is just a little bit shorter than some integer dividend of D_S.

Lemma 2.4.5: Let M be an m-stage digital system with a clocking scheme $(D_1, D_2, \ldots, D_m)$. On M, Let $\tau_t(x)$ be the execution time of a sequence of n micro cycles with a clock period x. For any execution sequence with $n_b > 0$ branches,

$$\lim_{e \to 0} \tau_t(\frac{D_S}{k+e}) > \tau_t(\frac{D_S}{k}) \text{ for any integer k, } \left\lceil \frac{D_S}{D_{max}} \right\rceil > k \geq 1, \text{ and real e.}$$

Proof: Let n_b be the number of branch micro cycles out of total n micro cycles to be executed. By Theorem 2.4.3,

$$\tau_t(\frac{D_S}{k+e}) = \{(n-n_b-1) + \lceil k+e \rceil \cdot (n_b + 1)\} \cdot \frac{D_S}{k+e} \text{ and} \qquad (1)$$

$$\mathcal{T}_t(\frac{D_S}{k}) = \{(n-n_b-1) + k\cdot(n_b + 1)\}\cdot\frac{D_S}{k} \quad (2)$$

From (1), $\lim_{e \to 0} \mathcal{T}_t(\frac{D_S}{k+e}) \approx^{12} \{(n-n_b-1) + (k+1)\cdot(n_b + 1)\}\cdot\frac{D_S}{k}$ (3)

By subtracting (2) from (3), we get

$$\lim_{e \to 0} \mathcal{T}_t(\frac{D_S}{k+e}) - \mathcal{T}_t(\frac{D_S}{k}) \approx \{(k+1) - k\}\cdot(n_b + 1)\cdot\frac{D_S}{k} = (n_b + 1)\cdot\frac{D_S}{k} > 0.$$

Therefore, $\lim_{e \to 0} \mathcal{T}_t(\frac{D_S}{k+e}) > \mathcal{T}_t(\frac{D_S}{k})$

□

With Lemmas 2.4.4 and 2.4.5, we can see that the execution time function in terms of the clock period is not linear and reducing the clock period does not always reduce the execution time. However, we can determine an optimal clock period of an m-stage system with a fixed clocking scheme with the following lemma:

Lemma 2.4.6: Let $\mathcal{M}$ be an m-stage digital system with a clocking scheme $(D_1, D_2, \ldots, D_m)$. Let $\mathcal{T}_t(x)$ be the execution time of a sequence of n micro cycles with a clock period x. On $\mathcal{M}$, for any execution sequence,

$$\min(\mathcal{T}_t) = \min\{\mathcal{T}_t(D_{max}), \mathcal{T}_t(\frac{D_S}{p-1})\}, \text{ where } p = \left\lceil \frac{D_S}{D_{max}} \right\rceil$$

Proof: $t_{cy} \geq D_{max}$ and, by the definition of p, $\frac{D_S}{p} \leq D_{max} \leq \frac{D_S}{p-1}$.

Accordingly we can partition the range of t_{cy} into $D_{max} \leq t_{cy} < \frac{D_S}{p-1}$ and $t_{cy} \geq \frac{D_S}{p-1}$. Then,

---

[12] Due to the discontinuity of the ceiling function, even when n_b is zero, Equations (1) and (2) are not equal.

$$\min(\mathcal{T}_t) = \min[\ \min\{\mathcal{T}_t(t_{cy}) \mid D_{max} \le t_{cy} < \frac{D_S}{p-1}\}\ ,\ \min\{\mathcal{T}_t(t_{cy}) \mid t_{cy} \ge \frac{D_S}{p-1}\}\ ]$$

(i) By Lemma 2.4.4, $\min\{\mathcal{T}_t(t_{cy}) \mid t_{cy} \ge \frac{D_S}{p-1}\} = \mathcal{T}_t(\frac{D_S}{p-1})$

(ii) $\frac{D_S}{p} \le D_{max} < \frac{D_S}{p-1}$.

From Theorem 2.4.3,

$$\mathcal{T}_t(t_{cy}) = (n - n_b - 1)\cdot t_{cy} + p\cdot(n_b + 1)\cdot t_{cy}, \text{ where } \frac{D_S}{p} \le t_{cy} < \frac{D_S}{p-1}. \qquad (1)$$

(1) is a linearly increasing function with the slope $(n - n_b - 1) + (n_b + 1)\cdot p > 0$.

Thus, $\min\{\mathcal{T}_t(t_{cy}) \mid D_{max} \le t_{cy} < \frac{D_S}{p-1}\} = \mathcal{T}_t(D_{max})$.

Therefore, according to (i) and (ii),

$$\min(\mathcal{T}_t) = \min\{\mathcal{T}_t(D_{max}),\ \mathcal{T}_t(\frac{D_S}{p-1})\}, \text{ where } p = \left\lceil \frac{D_S}{D_{max}} \right\rceil$$

□

Using Lemma 2.4.6, we can determine an optimal clock period by evaluating the execution time of the given execution sequence(s) for only two clock periods. In practice, execution sequences may be nondeterministic due to nondeterministic conditional branches (e.g., conditional branches on some external conditions and exception handling). However, if we can obtain statistics regarding the average length and composition of the execution sequence(s), then using Lemma 2.4.6, we can easily estimate an optimal clock period.

Theorem 2.4.7: Let $\mathcal{M}$ be an m-stage digital system with a clocking scheme $(D_1, D_2, \ldots, D_m)$. On $\mathcal{M}$, for any execution sequence,

$$\min(\tau_t) = \begin{cases} \tau_t(D_{max}) & \text{if } (n_b + 1)\cdot(D_{max} - l) < (n-n_b-1)\cdot\frac{l}{p-1} \\ \tau_t(D_{max}) = \tau_t(\frac{D_S}{p-1}) & \text{if } (n_b + 1)\cdot(D_{max} - l) = (n-n_b-1)\cdot\frac{l}{p-1} \\ \tau_t(\frac{D_S}{p-1}) & \text{if } (n_b + 1)\cdot(D_{max} - l) > (n-n_b-1)\cdot\frac{l}{p-1} \end{cases}$$

$$\text{where } p = \left\lceil \frac{D_S}{D_{max}} \right\rceil \text{ and } l = D_S - (p - 1)\cdot D_{max}$$

Proof: We prove the theorem by comparing $\tau_t(D_{max})$ and $\tau_t(\frac{D_S}{p-1})$. From Theorem 2.4.3, we know that:

$$\tau_t(D_{max}) = (n-n_b-1)\cdot D_{max} + (n_b + 1)\cdot p\cdot D_{max} \text{ and}$$

$$\tau_t(\tfrac{D_S}{p-1}) = (n-n_b-1)\cdot\tfrac{D_S}{p-1} + (n_b + 1)\cdot D_S$$

$$= (n-n_b-1)\cdot(D_{max} + \tfrac{l}{p-1}) + (n_b + 1)\cdot\{(p-1)\cdot D_{max} + l\}$$

$$\text{Thus, } \tau_t(D_{max}) - \tau_t(\tfrac{D_S}{p-1}) = (n-n_b-1)\cdot(-\tfrac{l}{p-1}) + (n_b + 1)\cdot(D_{max} - l) \qquad (1)$$

Therefore, from (1),

1. if $(n_b + 1)\cdot(D_{max} - l) < (n-n_b-1)\cdot\frac{l}{p-1}$, then $\tau_t(D_{max}) - \tau_t(\frac{D_S}{p-1}) < 0$ and therefore, $\tau_t(D_{max}) < \tau_t(\frac{D_S}{p-1})$
2. if $(n_b + 1)\cdot(D_{max} - l) = (n-n_b-1)\cdot\{l/(p - 1)\}$, then $\tau_t(D_{max}) = \tau_t(\frac{D_S}{p-1})$
3. if otherwise, $\tau_t(D_{max}) \geq \tau_t(\frac{D_S}{p-1})$.

□

Lemma 2.4.6 and Theorem 2.4.7 show that the shortest possible clock period, D_{max}, may not be optimal. Next, we will prove that any clocking scheme other than the original interstage propagation delays with $D_{max} > d_{max}$ (accordingly $D_S > d_S$) will always result in slower execution speed for the same execution sequence of micro cycles.

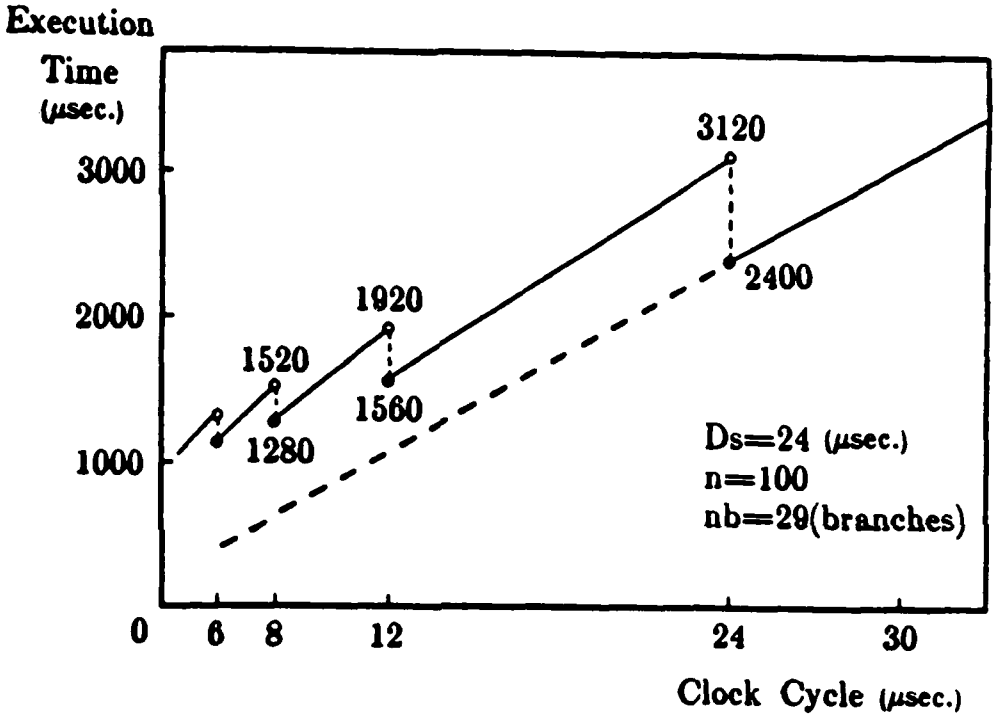


Figure 2.4-1: Execution time vs. clock period.

Figure 2.4-1 shows an example of the relationship between the execution time of an execution sequence and the chosen clock period. As shown in Figure 2.4-1, if there is branch micro cycle in the execution sequence, then execution time T is a discontinuous function of the clock period t_{cy}. The slope of each straight-line is determined by the number of branches (n_b). If there is no branch cycle in the execution sequence, then T becomes a straight line, as shown with a broken line.

Theorem 2.4.8: For an m-stage system, $\mathcal{M}$, with interstage propagation delays $(d_1, d_2, \ldots, d_m)$, let $\Psi_d = (d_1, d_2, \ldots, d_m)$ (same as the original minimum possible stage times) and $\Psi_D = (D_1, D_2, \ldots, D_m)$, where $D_i > d_i$, $1 \leq i \leq m$, be two different clocking schemes. Also let $\tau_t^{\Psi_D}$ and $\tau_t^{\Psi_d}$ be the execution times of a sequence of micro cycles with clocking sequences Ψ_D and Ψ_d, respectively.

If $D_{max} > d_{max}$ (then also $D_S > d_S$), then $\min(\tau_t^{\Psi_D}) > \min(\tau_t^{\Psi_d})$

Proof: By Theorem 2.4.3 and Lemma 2.4.4, we know that

$$\tau_t^{\Psi_D}(t_{cy}) = (n-n_b-1)\cdot t_{cy} + (n_b+1)\cdot\left\lceil\frac{D_S}{t_{cy}}\right\rceil\cdot t_{cy},\ t_{cy} \geq D_{max} \tag{1}$$

$$\tau_t^{\Psi_d}(t_{cy}) = (n-n_b-1)\cdot t_{cy} + (n_b+1)\cdot\left\lceil\frac{d_S}{t_{cy}}\right\rceil\cdot t_{cy},\ t_{cy} \geq d_{max} \tag{2}$$

$$\min(\tau_t^{\Psi_D}) = \min\{\tau_t^{\Psi_D}(D_{max}),\ \tau_t^{\Psi_D}(\tfrac{D_S}{p-1})\},\ \text{where } p = \left\lceil\frac{D_S}{D_{max}}\right\rceil \tag{3}$$

$$\min(\tau_t^{\Psi_d}) = \min\{\tau_t^{\Psi_d}(d_{max}),\ \tau_t^{\Psi_d}(\tfrac{d_S}{q-1})\},\ \text{where } q = \left\lceil\frac{d_S}{d_{max}}\right\rceil \tag{4}$$

Then, from (1) and (3),

$$\begin{aligned}\tau_t^{\Psi_D}(D_{max}) &= (n-n_b-1)\cdot D_{max} + (n_b+1)\cdot\left\lceil\frac{D_S}{D_{max}}\right\rceil\cdot D_{max}\\ &\geq (n-n_b-1)\cdot D_{max} + (n_b+1)\cdot\left\lceil\frac{d_S}{D_{max}}\right\rceil\cdot D_{max}\\ &\geq \tau_t^{\Psi_d}(D_{max})\end{aligned} \tag{5}$$

$$\begin{aligned}\tau_t^{\Psi_D}(\tfrac{D_S}{p-1}) &= (n-n_b-1)\cdot\frac{D_S}{p-1} + (n_b+1)\cdot D_S\\ &\geq (n-n_b-1)\cdot\frac{D_S}{p-1} + (n_b+1)\cdot\left\lceil\frac{d_S}{D_S}(p-1)\right\rceil\cdot\frac{D_S}{p-1}\\ &\geq \tau_t^{\Psi_d}(\tfrac{D_S}{p-1})\end{aligned} \tag{6}$$

CASE 1: If $\tau_t^{\Psi_d}(d_{max}) < \tau_t^{\Psi_d}(\frac{d_S}{q-1})$, then, obviously from (5) and (6) and by Lemma 6,

From (5), $\tau_t^{\Psi_D}(D_{max}) \geq \tau_t^{\Psi_d}(D_{max}) > \tau_t^{\Psi_d}(d_{max})$ and

from (6), $\tau_t^{\Psi_D}(\frac{D_S}{p-1}) > \tau_t^{\Psi_d}(\frac{D_S}{p-1}) > \tau_t^{\Psi_d}(d_{max})$

Thus, $\min(\tau_t^{\Psi_D}) > \min(\tau_t^{\Psi_d})$

CASE 2: If $\tau_t^{\Psi_d}(d_{max}) \geq \tau_t^{\Psi_d}(\frac{d_S}{q-1})$, then from (1) and (2),

$$\begin{aligned}
\tau_t^{\Psi_D}(\tfrac{d_S}{q-1}) &= (n-n_b-1)\cdot\tfrac{d_S}{q-1} + (n_b+1)\cdot\left\lceil\tfrac{D_S}{d_S}\cdot(q-1)\right\rceil\cdot\tfrac{d_S}{q-1} \\
&\geq (n-n_b-1)\cdot\tfrac{d_S}{q-1} + (n_b+1)\cdot q\cdot\tfrac{d_S}{q-1} \\
&> (n-n_b-1)\cdot\tfrac{d_S}{q-1} + (n_b+1)\cdot d_S \\
&> \tau_t^{\Psi_d}(\tfrac{d_S}{q-1})
\end{aligned}$$

Therefore, if $D_{max} > d_{max}$, $\min(\tau_t^{\Psi_D}) > \min(\tau_t^{\Psi_d})$ is always true. □

Theorem 2.4.8 shows that, even if the optimal clock period resulting in the fastest execution speed is chosen to be longer than the longest interstage propagation delay, increasing the longest stage time (D_{max}) will always result in a slower maximum execution speed. In other words, in Figure 2.4-1, if the longest stage time is increased, then the length of the clocking sequence (D_S) is also increased and the execution time curves are shifted upward and to the right.

Chapter 3

Synthesis of Maximum Execution Overlap Designs

3.1. Introduction

As we discussed in the previous chapter, if the average execution sequence of micro cycles is known, the performance of the maximum execution overlap scheme is determined by the clocking scheme used. However, the optimality of the clocking scheme is a function of the stage partitioning since the stage delays determine the minimum requirements of the clocking scheme. In order to choose an optimal number of stages and to determine whether to use a multistage scheme or not, we need the following:

1. a method to partition the system into a certain number, k, of stages which maximizes execution speed,

2. a method for performance comparison of a k-stage scheme to a j-stage scheme, $j \neq k$, with given system specifications and statistics regarding the execution sequence(s), and

3. a technique for cost analysis (including speed/cost tradeoff) of a multistage system compared to a single stage system.

We first discuss optimal stage partitioning algorithms. Then, we discuss performance comparison of different stage partitioning results. As an illustration of the techniques discussed, an example of stage partitioning of a microprogrammed CPU is given. Finally, we consider an extension of the technique to more general designs, which is demonstrated with a systolic array example. All the algorithms we discuss in this chapter have been programmed in Franz LISP and run on the VAX/750 under Berkeley UNIX 4.2.

3.2. Optimal Stage Partitioning

The optimal stage partitioning problem consists of three subproblems:

1. Given the maximum stage time limit (or a fixed clock cycle), partition every micro-cycle graph into a minimum number of stages so as to maximize performance and minimize latch cost.

2. Given the number of stages, k, partition the system into an optimal k-stage system to maximize execution speed (i.e., partition every micro-cycle graph in such a way that the number of stages is k while minimizing the longest stage time).

3. Determine the optimal number of stages, k, which maximizes the execution speed.

The third problem is a superset of the second problem, which in turn is a superset of the first. In order to partition the system into exactly k stages while minimizing the longest stage time, we need to partition the system with a number of different stage times and choose the best partition. After determining an optimal partitioning of the system for all possible cases of k, we need to compare the performance of a single stage scheme to a multistage scheme for certain k's. For this reason, we need efficient algorithms which can determine the optimal stage partitioning of a given design, given either the maximum stage time limit or the desired number of stages. In this section, we develop two optimal stage partitioning algorithms which run in polynomial time to the number of nodes in the input micro-cycle graphs.

3.2.1. Stage partitioning with a fixed stage time

The following procedure, KPART, partitions one or more micro-cycle graphs into the minimum number of partitions, k, necessary when the maximum stage time is limited to L_{max} (subproblem 1 above). Time delays due to the stage latches are also considered. Let δ_{max} be the longest module propagation delay. If $L_{max} = \delta_{max}$ + Dss + Dsp (absolute minimum possible

stage time), then k found by KPART is the minimum number of partitions to minimize the length of the longest partition of a given system.

The procedure, KPART, partitions each input micro-cycle graph into stages in such a way that, in every stage, no more nodes can be added without exceeding the maximum stage time limit while the data precedence of the original micro-cycle graphs is preserved. The procedure also determines the locations for the stage latches such that the data-flow moves from a stage only to its successor stage. The algorithm also computes the minimum possible clock period after the stage partitioning.

We first outline the algorithm, KPART, and then explain some important steps with an example shown in Figure 3.2-1. A more detailed description of this algorithm is in Appendix A together with the analysis of run-time complexity, which is O(|E|) where |E| is the total number of edges in the input micro-cycle graphs. The source code of an implementation in Franz LISP and a user's guide are found in [Parker 84].

Algorithm: KPART (stagetime-limit)

Variable

SF: Searching Fronts. The far-end nodes of the current partition.

NSF: Next Searching Fronts. The nodes to be the next SF.

NH: Next Heads. The starting SF for the next partition.

partition(i): The set of nodes in i-th partition

cutset(i): The set of edges for the output latches of the i-th partition (the (i+1)-th stage latch).

d(i): The propagation delay through the critical path of the i-th partition.

```
BEGIN {*kpart*}

1. Set k (partition index) = 1;

2. Get all the root nodes[13] and put in SF;

3. Add all the nodes in SF to partition(k);

4. FOR every node, i, in SF DO

     FOR every child, j, of i DO

       IF (every parent of j is in partition(x) for
           some x, x ≤ k, i.e. all its inputs are ready)

         THEN
           IF (including j does not violate
                the maximum stage time limit)
             THEN  put j in NSF;
             ELSE  put j in NH;

5. IF  not (empty NSF)
     THEN  move all the nodes in NSF to SF and go to 3;

     ELSE  compute d(k) for partition(k);
           go to 6;

6. IF  not (empty NH)
     THEN
         a. get every edge connecting a node in an already
            existing partition and a node which is not
            in any partition yet, and put it in cutset(i);
         b. move all the nodes in NH to SF;
         c. k := k + 1;
         d. go to 3;

     ELSE  get all the output edges for the output
           latch locations (the last stage latches);
           compute d(k);
           STOP;

END {*kpart*}
```

[13] Nodes with no parents.

partition is L_{max}. In order to have a smaller k, at least the length of one of the partitions must be increased and the boundaries be changed. Thus, at least one pair of v(i) and u(i+1) will be in the same partition, say P(i) (either P(i) or P(i+1)). Then,

1. If we move u(i+1) into P(i), then u(i) must not remain in P(i) in order not to increase the maximum length of the partition, L_{max}. Also, for the same reason, u(i) cannot be absorbed into P(i-1) without partitioning P(i-1).

2. Also, for the same reason, going the opposite direction, P(i) can only contain, at most, up to interval v(i+1).

Thus the number of stages remains, at least, the same. By repeating the adjustment according to the rules 1 and 2 until u(1) and v(k) are reached, we can see that the number of partitions cannot be decreased. Therefore, k is minimal.

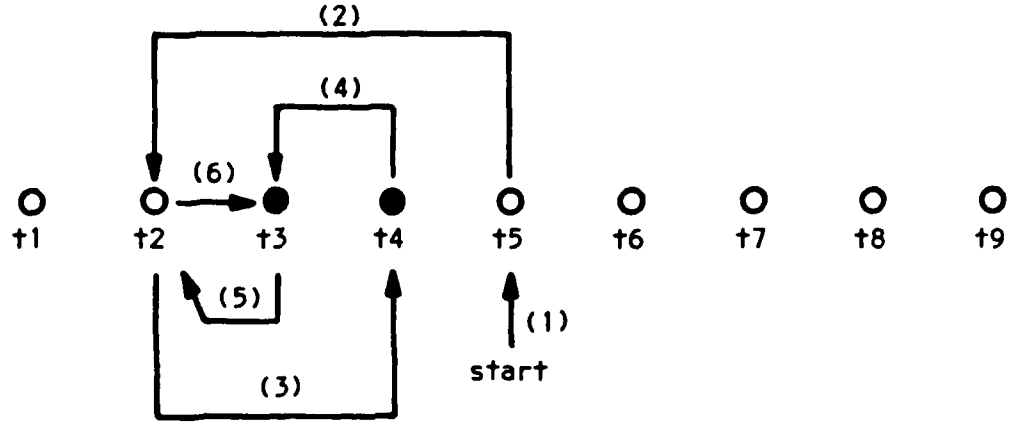
□

3.2.2. Stage partitioning with a fixed number of stages

Given a desired number of stages, K, we need to minimize the longest stage time, L_{max}. The following algorithm, OPART, first calls a procedure which enumerates the possible stage times of given micro-cycle graphs. It uses a *Mergesort* procedure to choose and sort all the distinct possible stage times. The stage partitioning algorithm, KPART, which we have discussed in the previous section, is called to check the feasibility of choices of L_{max} in a binary search fashion out of all possible stage times.

There may be more than one feasible stage time which result in the desired number of stages. Therefore, even after a feasible L_{max} is found, the binary search continues until the minimum feasible L_{max} is found to determine an optimal K-partition of a system with a given K. For example, suppose that

there are nine possible stage times, (t1 t2 ... t8 t9), sorted in non-decreasing order from left to right. Let t3 and t4 be the only stage times which partition the system into the exactly desired number of stages. The binary search proceeds as depicted below. The numbers in the parentheses represent the search order.



Algorithm *OPART*(K);

```
{*  K     ....  Input. Desired number of partitions          *}
{* dmax   ....  Output. Length of the longest partition     *}
{* p(i)   ....  Output. Locations for the i-th stage latches *}

   BEGIN {*OPART*}

      {*enumerate all possible stage times*}
      {*and sort in non-decreasing order  *}
      l[1..N] := findintervals;
      s[1..N] := Mergesort(l[1..N]);

      startpoint := ⌈N/2⌉

      level      := 1     {*depth in the binary search tree*}

      {* binary search among all possible stage times *}
      WHILE (startpoint ≠ lastpoint) DO
```

```
begin {*while*}

    {* compute the number of stages and the actual *}
    {* maximum stage propagation delay.            *}
    {*
    k, d_max <-- KPART(MCGs, s[startpoint], Dss, Dsp);

    {* update status *}
    level := level+1

    step := ⌈N/(2**level)⌉;

    {* determine the direction of next search *}
    IF k > K
        THEN
            startpoint := startpoint + step

        {* even if k = K, continue search *}
        ELSE
            if k = K then lastpoint := startpoint;
            startpoint := startpoint - step;
end {*while*}

Report the stage partitioning results with s[lastpoint];

end {*OPART*}
```

Run Time Analysis : Let |V| be the total number of nodes in the input micro-cycle graphs. The first step, finding all possible stage times, can be done by computing the longest path between every two connected nodes. The computation of the longest path from a node to every node in a directed acyclic graph takes O(|E|) time steps where |E| is the number of edges in the graph, using a dynamic programming algorithm [Horowitz 78]. If we repeat this for every node, the total number of time steps required will be O(|V||E|). Next, there can be at maximum $_{|V|}C_2$ pairs of connected nodes, and thus there are $O(|V|^2)$ number of possible stage times. The *MERGESORT* step for $O(|V|^2)$ elements takes $O(|V|^2 log|V|)$ time steps. Finally, the main loop is

iterated $O(log|V|^2 = log|V|)$ times (binary search). At each iteration of the main loop, the procedure, KPART, is called once, which takes $O(|E|)$ time steps. Therefore, the total execution time of the main loop is $O(|E|log|V|)$. Thus, the time complexity of the algorithm, OPART, is determined by the *Mergesort*, and is $O(|V|^2 log|V|)$.

Lemma 9 : d_{max} computed by the algorithm *OPART* is minimal.

Proof : Proof is obvious by the construction of the algorithm and its procedures. s[i]'s are the only possible cases of the length of any partition and the algorithm chooses the minimum possible length from s, using the binary search as depicted above. Therefore, d_{max} is minimal.

□

3.3. Performance Comparison - K Stage vs. Single Stage

As the number of branch executions increases, the efficiency of a multistage system decreases due to the additional delay through the interstage latches. Also, if the longest interstage propagation delay (D_{max}) is too long, the performance of a multistage system may not be as good as a single stage system since the amount of overlapped execution time may be very small. Using the execution time equations developed in Sections 4.2 and 4.3, we must compare the average expected execution speeds of the possible configurations of the system. That is, we must compare

$T = \{n_d + \left\lceil \frac{D_S}{t_{cy}} \right\rceil \cdot (n_b + 1) \} \cdot t_{cy}$ of multistage configurations and

$T = n \cdot t'_{cy}$ of a non-overlapped configuration.

where t'_{cy} is the critical path propagation delay of the micro-cycle graphs.

In addition, we must consider the cost increase. A multistage implementation of a system requires some additional hardware such as interstage latches and a multiphase clock generator. Routing the multiphase clock signals may cause problems in the same way that power routing does.

3.4. An Example Maximum Overlap Scheme For a Microprogrammed CPU

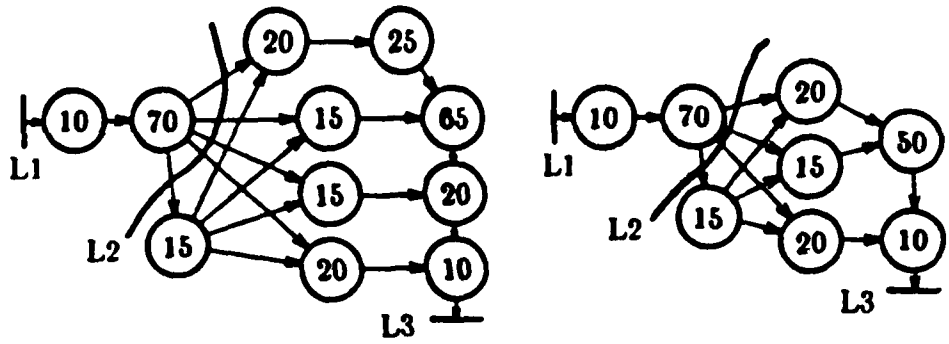
In this section, we demonstrate the results of static clocking scheme synthesis through an example of a microprogrammed CPU, the HP-21MX. This example shows how the proposed technique can be used to complete a partial design.

The circuit graph of the HP-21MX CPU is shown in Figure 2.3-1 and the corresponding micro-cycle graphs are shown in Figure 2.3-2.

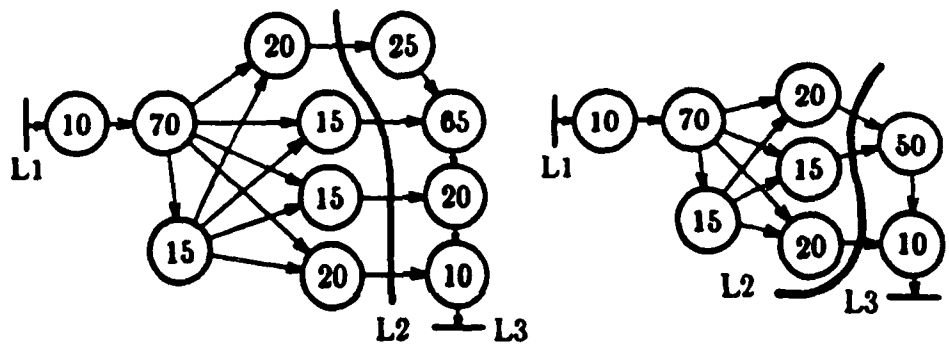
Three different results of stage partitionings are shown in Figure 3.4-1. (a) is the original 3-stage configuration used. (b) and (c) show the optimal 3-stage and 4-stage partitionings determined by the algorithm OPART.

We assume that Dss is 5 nsec. and Dsp is 10 nsec. The 3-stage partition (b) is obtained when $L_{max} = 130$ nsec. including 15 nsec. total for Dss and Dsp of the stage latches. The 4-stage partition (c) is obtained when $L_{max} = 110$ also including 15 nsec. total for Dss and Dsp.

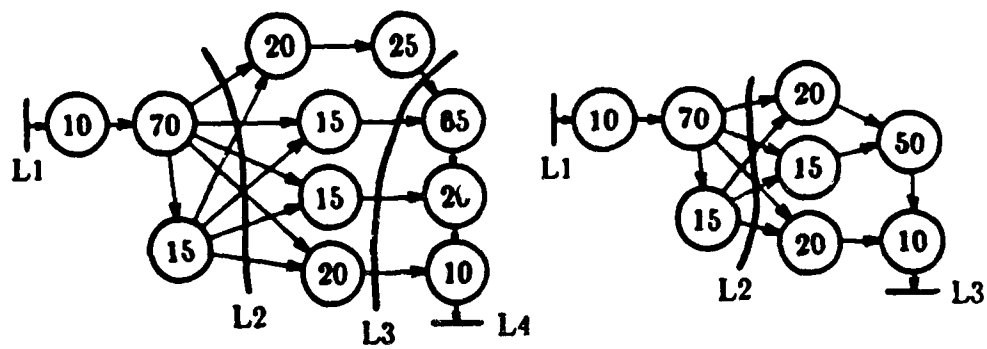
The timing values determined by the stage partitionings are listed below. The lengths of the clock phases have a certain safety margin, as shown.



(a) The Original 3-Stage Configuration(Lmax=175nsec.)



(b) Another 3-Stage Configuration(Lmax=135nsec.)

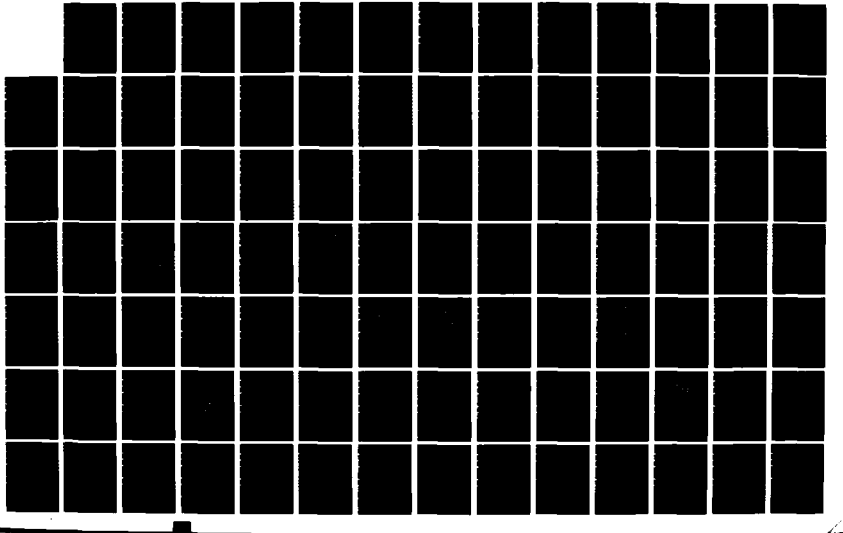


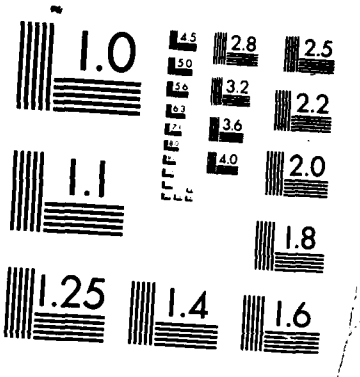
(c) A 4-Stage Configuration(Lmax=115nsec.)

Figure 3.4-1: Stage partitioning of the HP-21MX CPU.

AD-A164 690 SYNTHESIS OF HIGH-SPEED DIGITAL SYSTEMS(U) UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES COMPUTER RESEARCH INST N PARK 08 NOV 85 CRI-85-23 ARO-20637.11-EL 2/3

UNCLASSIFIED DAAG29-83-K-0147 F/G 9/2 NL

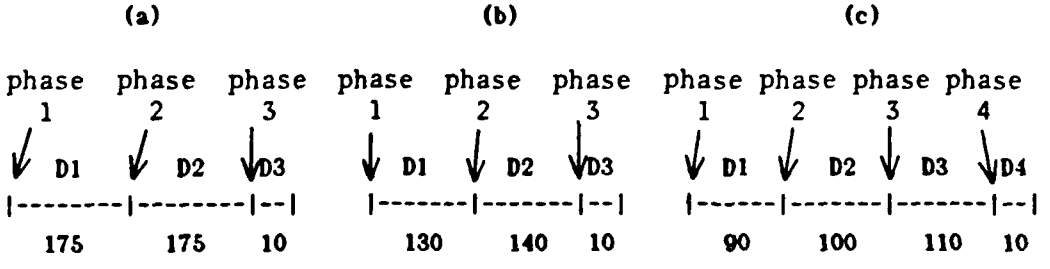




(a) The original 3-stage scheme
(b) A new 3-stage scheme
(c) A 4-stage scheme

partition	(a)	(b)	(c)	
d_{max}	165	130	105	nsec.
D_{max} (t_{cy})	175	140	110	nsec.
D_1	175	130	90	nsec.
D_2	175	140	100	nsec.
D_3	10	10	110	nsec.
D_4	-	-	10	nsec.
D_S	360	280	310	nsec.
$\|L_1\|$	24	26	58	bits

The corresponding clocking sequences are shown below.



For configurations (a) and (b), there is no resynchronization overhead. For configuration (c), there may be data contention between two minor cycles, the "store result (D4)" of a micro cycle and the "read operand (D2)" of the next micro cycle, which requires delay of the next micro cycle fetch for one

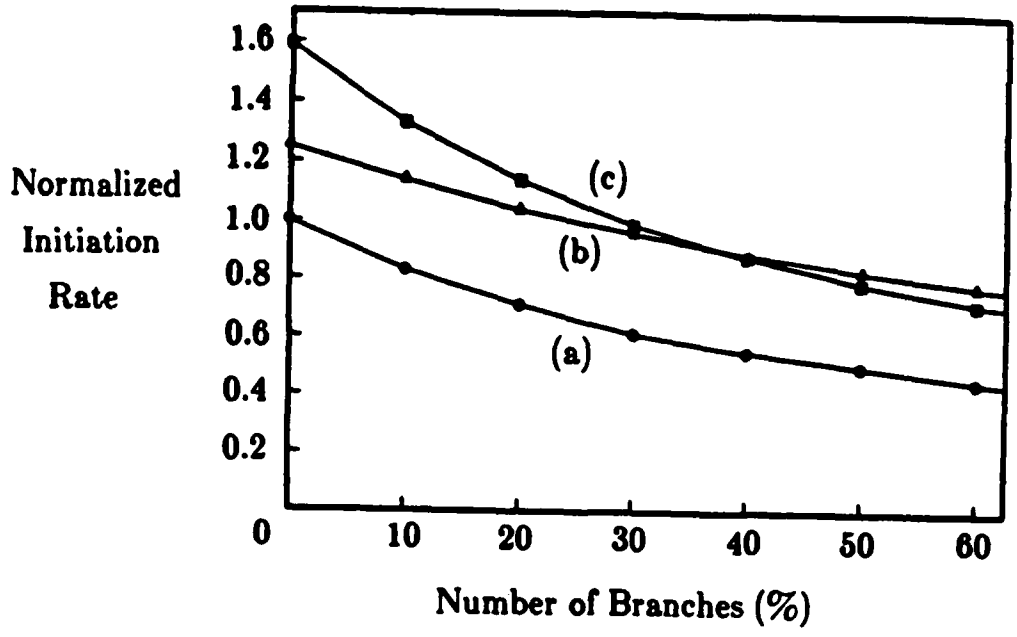
clock period. The branching overhead of configurations (a) and (c)[14] is two clock periods. For configuration (b), the branching overhead is only one clock period since $\left\lceil \frac{D_S}{t_{cy}} \right\rceil - 1 = 1$ (Lemma 2.4.2).

We first compare the original design (a) and our 3-stage partitioning (b). As shown in the circuit graph of Figure 2.3-1, the second stage latch of the original design is the micro-instruction buffer, which is usually determined in *ad hoc* fashion and most widely used in microprogrammed controller designs. However, as shown in Figure 3.4-2, we increase the performance of the system significantly by moving the location of the second latch. The cost increase is only 2 latch bits.

The performance comparison of the three configurations is plotted in Figure 3.4-2. For each configuration, the execution times for 100 micro cycles are computed with different numbers of branch cycles and resynchronizations. As shown in the figure, the 4-stage configuration shows the best performance in general. In the worst cases when more than half of the micro cycles either branch or need resynchronization, the performance of the 4-stage configuration, (c), is worse than that of (b). However, such cases are unusual. In such cases, we can re-compute the optimal clock period and corresponding execution time using Theorem 2.4.7 to determine whether to use the multistage scheme or not.

---

[14]Refer to Figure 3.4-1 and Equation (2.4.3) for the calculation of the branching overheads. For all the configurations, we assume that the lengths of the clock phases are fixed and no wait clock periods are added.



(a) The original 3-stage scheme
(b) A new 3-stage scheme
(c) A 4-stage scheme

Figure 3.4-2: Performance comparison of the HP-21MX CPU.

3.5. Extensions to More General Digital Designs

3.5.1. Execution overlap schemes for completed designs

The synthesis techniques for maximum execution overlap we have discussed so far can also be applied to some specific designs with an existing clocking scheme. The technique we will discuss in this section can be applied to general cases of digital designs which might even have loops (i.e., use the same module more than once) and have registers to store values which are used by a sequence of computation tasks.

Figure 3.5-1-(a) indicates a circuit which is completed with a clocking scheme. Suppose that we can partition the circuit into blocks such that there is no data or control flow in both directions between any two blocks. An example partitioning of the circuit of Figure 3.5-1 into two blocks is shown in Figure 3.5-1-(b). As long as such a partition exists, we can overlap the execution of a circuit. This is very similar to pipelines with pipelined stages.

Conjecture 3.1: For any circuit that can be partitioned into a chain of blocks without data or control flow in both directions between any two blocks, execution overlap is possible between blocks.

Outline of Proof: As long as there is no bidirectional data or control flow between two blocks, only a predecessor affects the execution of its successor. If we put a latch and store the data or control values produced by a block which are needed for its successor, the successor can start execution while the predecessor either completes the same task or starts execution of a new task.

In a partitioned circuit as such, we can consider the blocks to be the nodes in the micro-cycle graphs and apply the same techniques we have discussed for maximum overlap. However, since a block may use clocks during

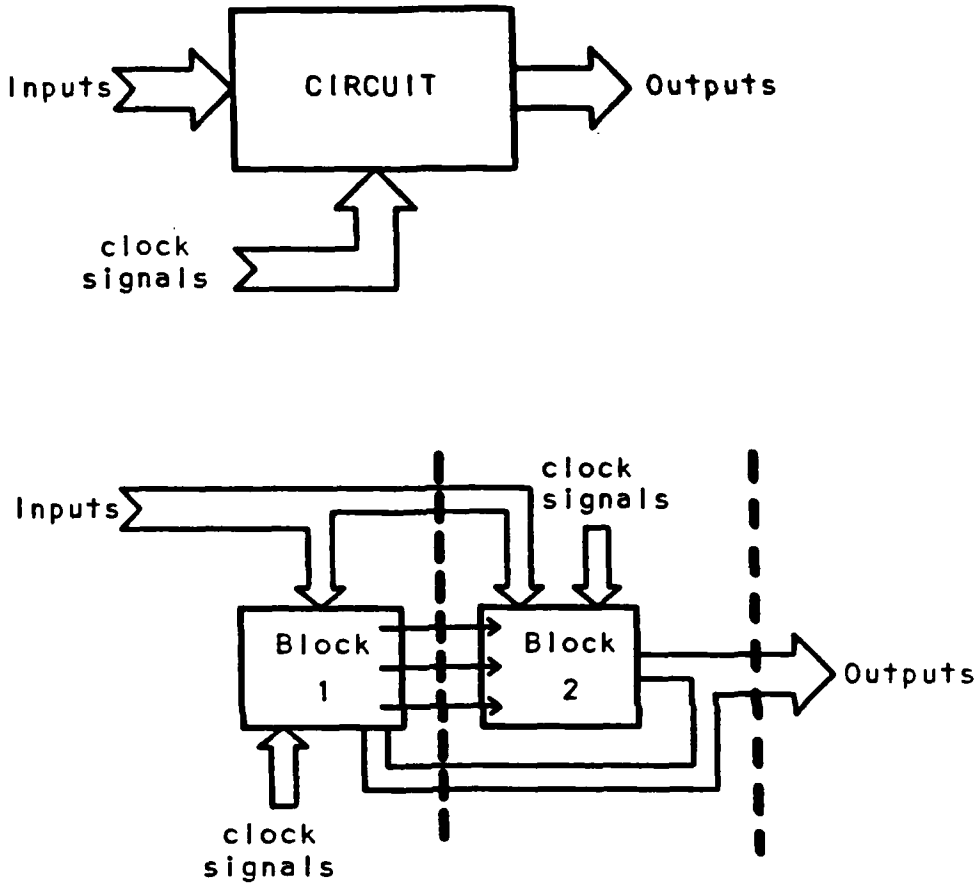


Figure 3.5-1: An example of circuit partitioning.

its execution, the delay time of a block must be computed as the total execution time taken by it. The actual amount of overlap is determined by the execution time of each block. If the execution time of a block after partitioning is still the same as the total execution time before partitioning, there will be no gain in execution speed. An example of this case is when a block has a loop which is iterated many times to produce output even after the data needed for the next block is produced and the succeding blocks complete execution.

3.5.2. A systolic array example

In this example, we show how an already designed systolic array can be sped-up without changing the original data and control flow.

A systolic array design taken from [Leiserson 83] is shown in Figure 3.5-2, which continuously evaluates the function $y_i = \sum_{j=0}^{3} \delta(x_{i-j}, a_j)$. In the original design, the propagation delays of the registers are assumed to be negligible and we make the same assumption here. The clock period is 13, which is determined by the critical path $\delta 2 \rightarrow \delta 3 \rightarrow +1$. Each y_i is calculated by clocking all the registers R1 through R5 at the same time.

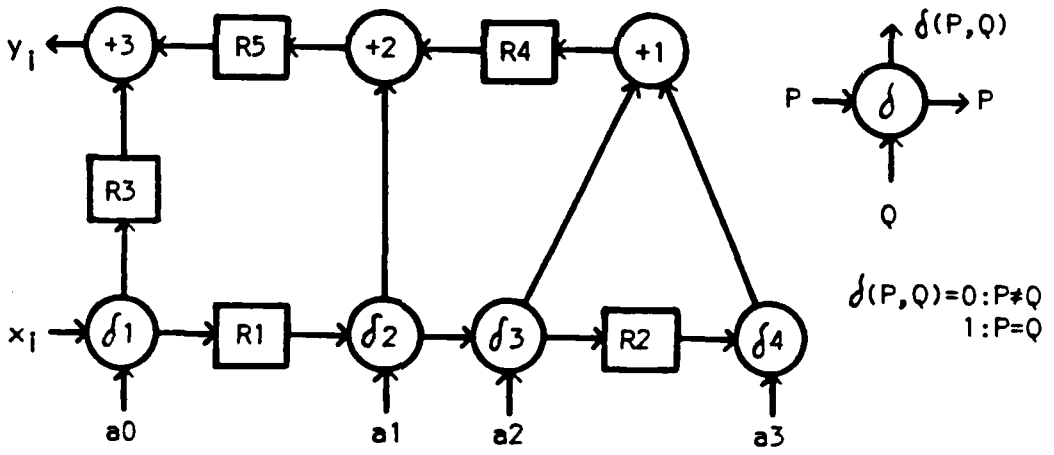


Figure 3.5-2: A systolic array evaluating $\sum_{j=0}^{3} \delta(x_{i-j}, a_j)$.

The goal of stage partitioning is to minimize the delay time between any two registers by inserting latches, i.e., to minimize the processing time in each block while maintaining the original data flow.

Figure 3.5-3 shows the results of stage partitioning. The desired interstage propagation delay is chosen to be the same as the longest module propagation delay, which is 7. As shown with the dashed cut line, four latches are to be inserted in between R3 and +3, $\delta 2$ and +2, $\delta 3$ and +1, and $\delta 4$ and +1 as the result of the partitioning.

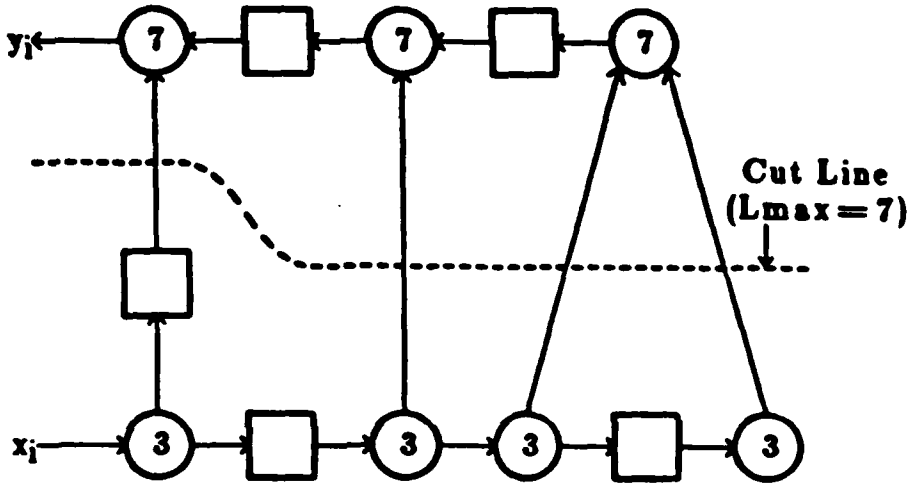


Figure 3.5-3: Stage partitioning of the systolic array of Fig. 3.5-2.

Figure 3.5-4 shows the modified systolic array after the registers, R6 through R9, are inserted. In this modified design, all the registers from R1 through R9 are clocked at the same time by a single clock source. In Table 3.5-1, the value flow through the registers in the original design and in the modified design are compared. Note that in the modified design the output is delayed by one clock cycle than in the original design. This is due to the fact that the inserted registers, R6 through R9, perform the role of the stage latches for execution overlap which delays the values through them by one clock cycle.

As shown in Figure 3.5-4, any path without passing through a latch is no

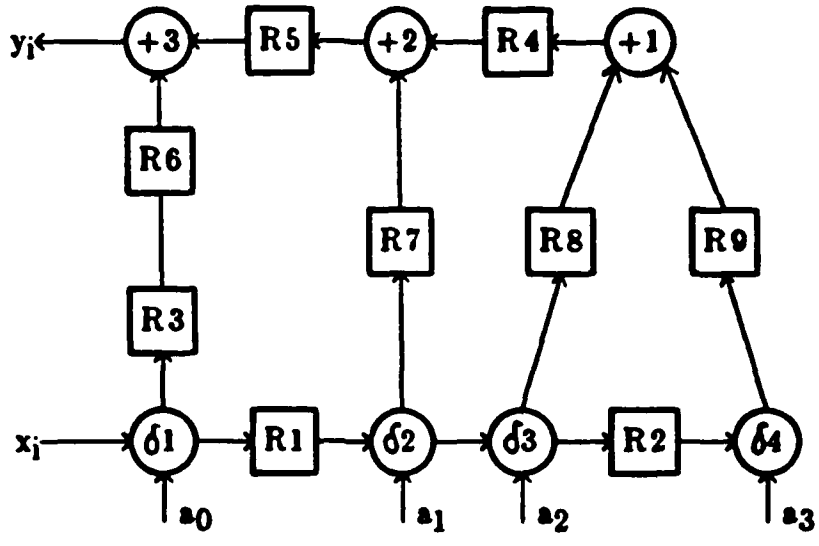


Figure 3.5-4: Stage partitioning result of the systolic array of Fig. 3.5-2.

longer than 7, thus, the resulting minimum possible clock period is 7. The systolic array continuously evaluates the same function every cycle and there are neither branch nor resynchronization overheads. Accordingly, the throughput rate is inversely proportional to the clock period. Therefore, the throughput rate is increased by (13-7)/7 = 85.7 (%). This throughput rate increase is achieved at the cost of the four added overlap stage latches.

Clock cycle	(a) Value flow in the original design. t_i	t_{i+1}	t_{i+2}	(b) Value flow in the modified design. t_i	t_{i+1}	t_{i+2}
R1	x_i	x_{i+1}	x_{i+2}	x_i	x_{i+1}	x_{i+2}
R2	x_{i-1}	x_i	x_{i+1}	x_{i-1}	x_i	x_{i+1}
R3	$\delta(x_i,a_0)$	$\delta(x_{i+1},a_0)$	$\delta(x_{i+2},a_0)$	$\delta(x_i,a_0)$	$\delta(x_{i+1},a_0)$	$\delta(x_{i+2},a_0)$
R4	$\delta(x_{i-1},a_2)$ $+\delta(x_{i-2},a_3)$	$\delta(x_i,a_2)$ $+\delta(x_{i-1},a_3)$	$\delta(x_{i+1},a_2)$ $+\delta(x_i,a_3)$	$\delta(x_{i-2},a_2)$ $+\delta(x_{i-3},a_3)$	$\delta(x_{i-1},a_2)$ $+\delta(x_{i-2},a_3)$	$\delta(x_i,a_2)$ $+\delta(x_{i-1},a_3)$
R5	$\delta(x_{i-1},a_1)$ $+\delta(x_{i-2},a_2)$ $+\delta(x_{i-3},a_3)$	$\delta(x_i,a_1)$ $+\delta(x_{i-1},a_2)$ $+\delta(x_{i-2},a_3)$	$\delta(x_{i+1},a_1)$ $+\delta(x_i,a_2)$ $+\delta(x_{i-1},a_3)$	$\delta(x_{i-2},a_1)$ $+\delta(x_{i-3},a_2)$ $+\delta(x_{i-4},a_3)$	$\delta(x_{i-1},a_1)$ $+\delta(x_{i-2},a_2)$ $+\delta(x_{i-3},a_3)$	$\delta(x_i,a_1)$ $+\delta(x_{i-1},a_2)$ $+\delta(x_{i-2},a_3)$
R6				$\delta(x_{i-1},a_0)$	$\delta(x_i,a_0)$	$\delta(x_{i+1},a_0)$
R7				$\delta(x_{i-1},a_1)$	$\delta(x_i,a_1)$	$\delta(x_{i+1},a_1)$
R8				$\delta(x_{i-1},a_2)$	$\delta(x_i,a_2)$	$\delta(x_{i+1},a_2)$
R9				$\delta(x_{i-2},a_3)$	$\delta(x_{i-1},a_3)$	$\delta(x_i,a_3)$
output	$\delta(x_{i-1},a_0)$ $+\delta(x_{i-2},a_1)$ $+\delta(x_{i-3},a_2)$ $+\delta(x_{i-4},a_3)$ $=y_{i-1}$	$\delta(x_i,a_0)$ $+\delta(x_{i-1},a_1)$ $+\delta(x_{i-2},a_2)$ $+\delta(x_{i-3},a_3)$ $=y_i$	$\delta(x_{i+1},a_0)$ $+\delta(x_i,a_1)$ $+\delta(x_{i-1},a_2)$ $+\delta(x_{i-2},a_3)$ $=y_{i+1}$	$\delta(x_{i-2},a_0)$ $+\delta(x_{i-3},a_1)$ $+\delta(x_{i-4},a_2)$ $+\delta(x_{i-5},a_3)$ $=y_{i-2}$	$\delta(x_{i-1},a_0)$ $+\delta(x_{i-2},a_1)$ $+\delta(x_{i-3},a_2)$ $+\delta(x_{i-4},a_3)$ $=y_{i-1}$	$\delta(x_i,a_0)$ $+\delta(x_{i-1},a_1)$ $+\delta(x_{i-2},a_2)$ $+\delta(x_{i-3},a_3)$ $=y_i$

Note: The output value is assumed to be latched to some external latch (or memory) at the same clock tick clocking the registers R1 through R5.

Table 3.5-1: The value flow in the systolic arrays of Fig. 3.5-2 and Fig. 3.5-4.

Chapter 4
Theory of Pipeline Synthesis

4.1. Introduction

Pipelining has been a good methodology for designing fast digital circuits. In this chapter, we analyze the characteristics and problems of automated synthesis of near-optimal (the cheapest and fastest) pipelines from a functional (behavioral) design description.

4.1.1. Pipelining

In pipelining, each unit computation task (e.g. a microinstruction) is partitioned into a sequence of subtasks and each subtask is executed during a clock cycle.[15] Every clock cycle is of the same time period. Consecutive tasks are initiated at some fixed or variable intervals (called **latency**), which are integer multiples of a clock cycle and are shorter than the execution time of a task. In this fashion, subtasks of consecutive tasks are executed overlapped in time on different parts of a pipeline circuit as in the maximum overlapped execution we discussed in Chapter 2.

Figure 4.1-1-(a) shows a data flow graph of a computation task partitioned into a sequence of five subtasks, F1 through F5. Subgraph Gi corresponds to the data-flow of Fi.

In pipelining, each subtask is executed during a single clock cycle. Figure

[15] A task corresponds to a micro task and a clock cycle corresponds to a minor cycle defined in Chapter 2.

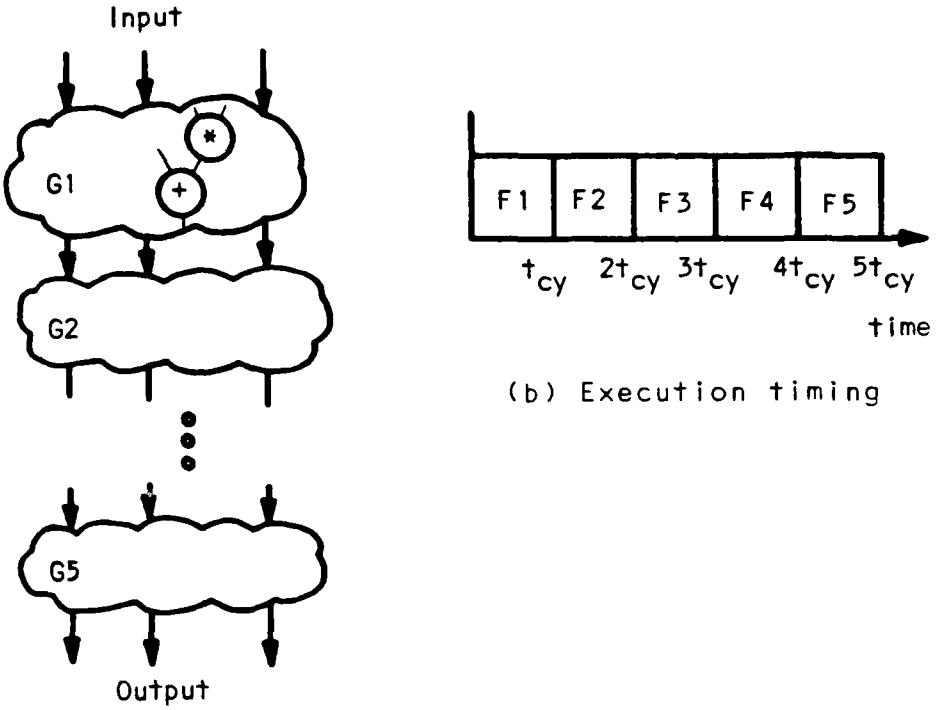


Figure 4.1-1: A partitioned data flow graph and its execution timing.

4.1-1-(b) shows the execution timing of a task. Figure 4.1-2-(a) shows an example of pipelined execution of a sequence of five tasks, I1 through I5.

Figure 4.1-2-(b) shows a 5-stage pipeline where the i-th stage executes subtask Fi. Depending on the timing specification/requirements of the input/output values, either the inputs or the outputs of each stage are to be latched internally so that *the computation of a subtask neither affects nor is affected by the computation of any other subtasks.*[16] In Figure 4.1-2-(b), the pipeline is filled with the subtasks of tasks I1 through I5. Stage S1 is executing F1 of I5, S2 is executing F2 of I4, and so on.

---

[16]The input or output latches perform the same role as the stage latches in maximum execution overlap.

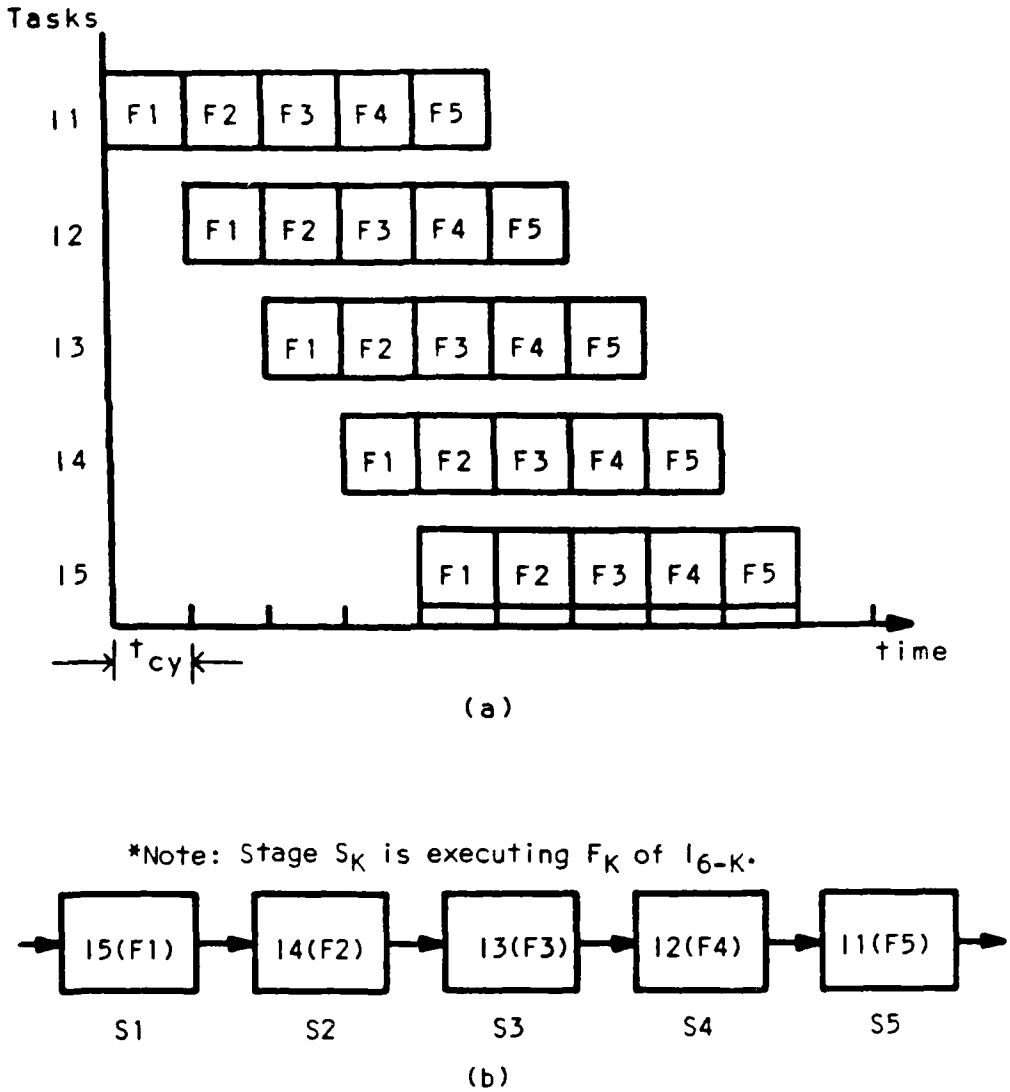
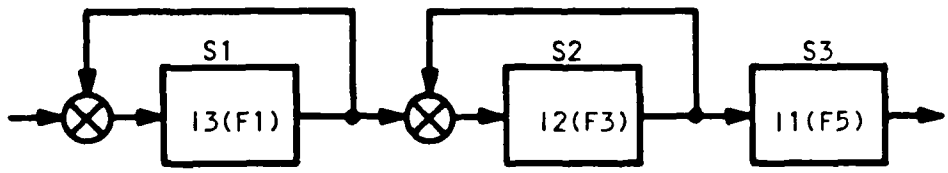


Figure 4.1-2: (a) A pipelining schedule and (b) a pipeline.

For the partitioned data flow graph of Figure 4.1-1, the fastest design would be one as shown in Figure 4.1-2. In this design, there is no resource sharing between the executions of subtasks, i.e., no subtask of a task needs the same stage more than once. However, if the total cost for the design is limited and not enough to allocate a different set of resources to each subtask, some resources must be shared by more than one subtask. Of course, resource

sharing will sacrifice performance. Figure 4.1-3 shows a slower but cheaper pipeline design.



Note: The pipeline is currently executing I1, I2, and I3. During the next clock cycle, S1 will execute F2 of I3, and S2 will execute F4 of I2.

Figure 4.1-3: A cheaper pipeline design.

In this design, stages S1 and S2 are reused. Stage S1 performs subtasks F1 and F2, and stage S2 performs F3 and F4. Figure 4.1-4 shows the timing diagram for the execution of consecutive tasks on this pipeline. As shown in this timing diagram, the initiation interval of this pipeline is doubled compared to the pipeline design of Figure 4.1-2.

4.1.2. Synthesis of pipelines at the functional level

The task of pipeline synthesis at functional level is that of producing a register-transfer level implementation of a pipeline from the functional (behavioral) description of the task. In this thesis, the input functional description is given as a data flow graph of the computation tasks to be executed. Design constraints are the total available cost budget and the minimum required performance. The design tasks include:

1. **scheduling** - assignment of operations to time steps or clock cycles, i.e., partitioning of a task into subtasks,

2. **resource allocation** - allocation of resources (how many of which types of modules) to subtasks, and

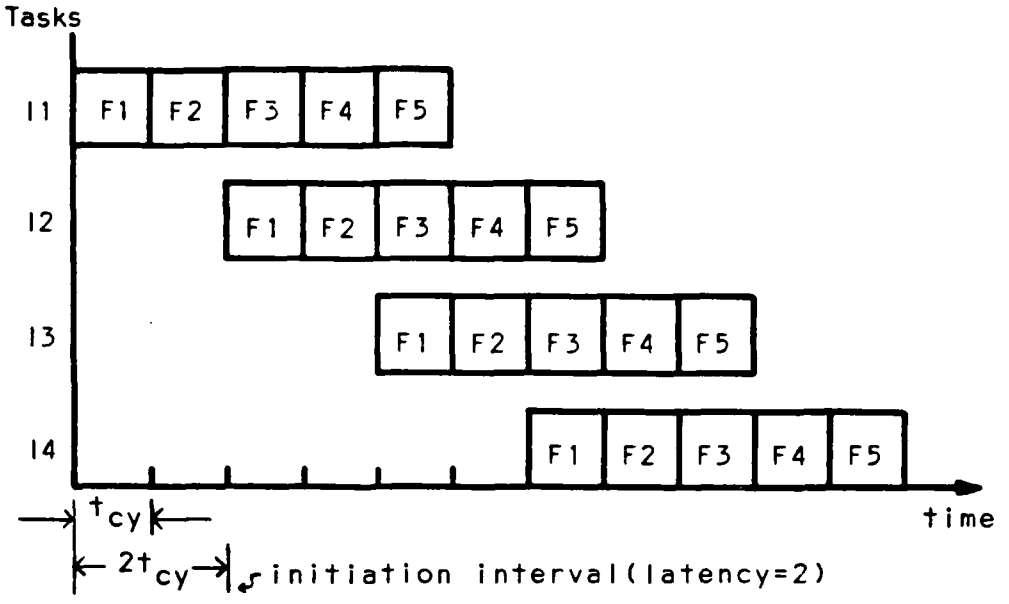


Figure 4.1-4: An example of pipelining with a fixed latency of 2.

3. **register-transfer synthesis** - detailed assignment of operations to operators for each subtask, and placement and interconnection of storage elements and multiplexers.

An optimal clocking scheme is a result of the execution of these three tasks.

A desired optimization goal is either maximizing performance within the cost constraint or minimizing total cost while satisfying the minimum required performance.

When an optimal design is desired, tasks 1, 2, and 3 cannot be performed separately. For example, the shortest schedule does not always guarantee the fastest performance since it might force the resource allocation in such a way that the fastest possible initiation rate of the tasks is not feasible due to resource conflicts between consecutive tasks. Also, an expensive pipeline does not always guarantee better performance since the scheduling may not be able

to utilize all the available resources. Finally, the feasibility of a resource allocation within cost constraints can only be known after the register-transfer synthesis is done.

As we have discussed in Section 1.3, there is no previous work reported on synthesis of pipelines at the functional level. Most of the previous work done in this area focused on the usage of already designed pipelines [Shar 72, Davidson 75, Patel 76, Kogge 81]. Some reported synthesis research focused on timing and control issues for gate-level synthesis of pipelines [Cotten 65, Kogge 81]. There has not been any attempt at automated synthesis at any level of pipeline design.

In this chapter, we aim for near-optimal automated pipeline synthesis at the functional level, considering all the cost and performance parameters of the expected usage of the pipeline. We develop a method for estimating the cost and performance of the pipeline design even before we actually synthesize the pipeline. This is important since the decision whether to use a pipeline (how beneficial it will be, and how well it will fit into the rest of the system) can be made without actually synthesizing one.

4.1.3. Basic terms

Some of the terms appearing in this chapter have already been defined and used in Chapter 2. However, for clarity of discussion, we summarize the basic terms which will be used most frequently in this chapter.

Data flow graph

A directed acyclic graph (DAG), G=(V,E), where the set of nodes, V, represents the operations on the values and the set of directed edges, E, represents the value flow (data precedence) between operations.[17]

---

[17] In fact, since an operation can only start execution when all its input values are ready, the arcs also represent the execution order or data precedence of the operations.

Stage latch A set of storage elements storing either the inputs or the outputs of a pipeline stage.

Stage time The minimum required time period for a stage to read input, complete execution, and latch output.

Clock cycle The time interval in which all the stage latches are clocked. This is the time period allocated to every stage to complete execution, and must be longer than the longest stage time.

Initiation interval (cycle)
The time interval at which a new task can be shifted into the pipeline, which is some integer multiple of the clock cycle.

Latency (Initiation interval)/(Clock cycle)

Fixed latency Latency which is a fixed constant. A new task can be initiated every *latency* clock ticks.

Variable latency
Latency which is a repeated sequence of integers. Tasks can be initiated in the time intervals as specified in a sequence, which is repeated. For example, a variable latency (2 3) can initiate tasks at the clock ticks of 0, 2, 5, 7, 10, 12, 15, ... and so on.

A task A unit computation which requires a pass through the pipe. This corresponds to a micro task defined in Chapter 2.

A subtask A partition of a task which is executed by a stage in the pipe. This corresponds to a minor task defined in Chapter 2.

Pipe cycle The time period for a task to be completed, i.e., the time period a task stays in the pipe (latency of the pipe).

4.1.4. A fixed latency assumption

As we have discussed before, there are two types of latencies used to initiate consecutive tasks, a fixed latency and a variable latency. *For synthesis, only a minimum possible fixed latency should be considered in order to optimize both the performance and the control circuitry.* As discussed and explained in the previous pipeline control work [Patel 76, Kogge 81], variable latency control requires a complex initiation scheduling mechanism in order not to cause any resource conflicts between consecutive tasks. In cases when a pipeline has 10 or more stages, the control mechanism for variable latency becomes impractical due to the complexity of the control circuitry and its processing time, which might be comparable to the initiation interval.

In fact, the maximum possible performance does not depend on the type of latency used but on the pattern of the resource usage, as we have discussed in Section 1.3. As we will discuss later in this chapter, for a fixed task, a fixed latency scheme can always achieve the maximum possible performance with proper scheduling and resource allocation. Therefore, variable latency control must be considered only when a fixed latency is not feasible due to the necessity of executing a different set of tasks on an existing pipeline.

For this reason, we only consider a fixed latency initiation of tasks.

4.2. A Data Flow Graph Model for Pipeline Synthesis

Prior to discussing the actual synthesis procedure, it is necessary to develop in some detail the data flow graph model which will be used to describe the behavior of the pipeline to be synthesized.

The data flow graph model used in this thesis is a modified subset of the *behavioral subspace* representation of the *Design Data Structure* (DDS) [Knapp 83, Knapp 85] of the USC Advanced Design AutoMation (ADAM) system [Granacki 85]. It is chosen in order to be compatible with other parts of the USC ADAM system. The ADAM DDS is designed for the representation of virtually almost all digital design cases. Since we focus on pipeline synthesis, we do not require all the general, complex constructs of the DDS in our input data-flow representation. We limit the types of constructs to be used and put restrictions on some of the chosen constructs. We first briefly review some aspects of the behavioral subspace representation of the ADAM DDS which are relevant to pipeline synthesis. A detailed description of the behavioral subspace representation is found in [Knapp 83, Knapp 85]. Next, we discuss the restrictions and limitations on the behavioral subspace representation for the construction of a data flow graph as a design description for pipeline synthesis.

4.2.1. A brief review of the behavioral subspace

A behavioral subspace representation of a design is essentially a data flow graph. However, since it is designed for the representation of digital hardware design (both design specification and design description), it has subtle differences from computer program or instruction data-flow descriptions [Dennis 74, Davis 82]. The main features of the behavioral subspace representation can be briefly summarized as follows:

Acyclic, single-assignment data flow: It is an acyclic graph (without any

loops). In fact, loops are unrolled by indexing input and output values of iterated operations. In a strict sense, it is not a pure single assignment representation as far as loops are concerned. An example of an unrolled loop is shown in Figure 4.2-1.

```
{* X has never been *}
{* written before   *}

For i := 0 to 2 DO
      X := X + a[i];
```

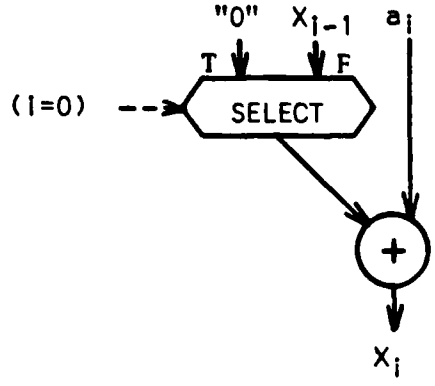


Figure 4.2-1: An example of loop unrolling with value indexing.

No control or timing information: It has no control or timing information such as tokens [Dennis 74] and explicit loop constructs [Davis 82, Snow 78]. Control and timing information is stored in the *timing subspace* of the ADAM Design Data Structure and can be looked-up through the *bindings* between the two subspaces. For the example data flow graph of Figure 4.2-1, the loop iteration count, 3, is found either in the control subspace or on the bindings between the behavioral and timing subspaces.

Implicit conditionals: Conditionals (conditional operations and values produced by conditional operations) are implicit and can only be explicitly referenced through the bindings to the timing subspace. In other words, in order to check whether a certain operation or value is conditional, either the whole data flow graph must be examined or the timing subspace must be looked-up. Figure 4.2-2 shows an example of implicit conditionals.

In figure 4.2-2, the subgraph G2 produces both unconditional results (O3)

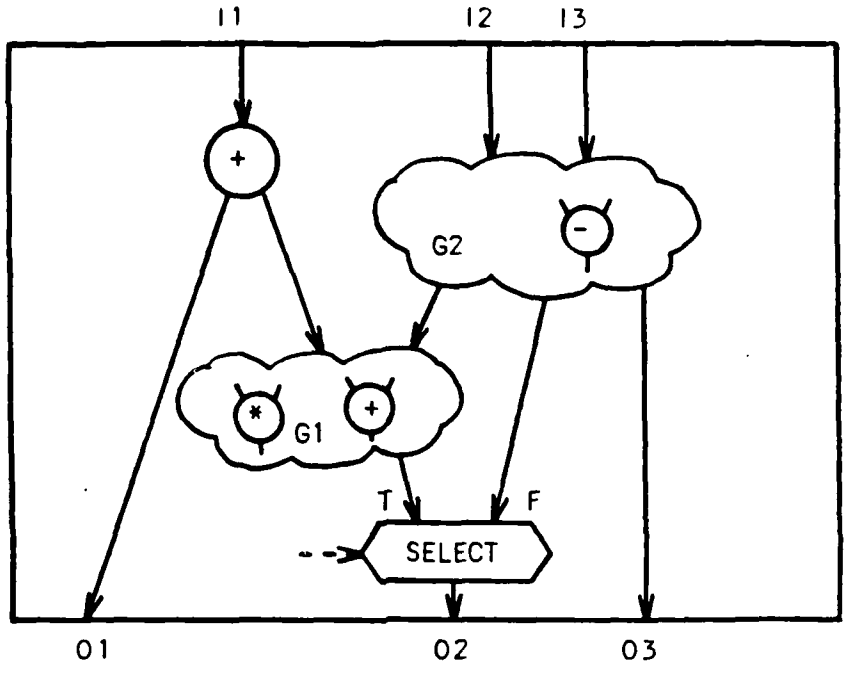


Figure 4.2-2: An example data flow graph with implicit conditionals.

and values which may not be needed depending on the condition input to the select node. In order to determine whether some operation node in G2 produces definitely needed values, we need to trace every possible execution path from the node to the output ports (O1, O2, and O3). If there is any path reaching to one of the output ports without going through any select node, then the operation must be performed regardless of any condition. This is true in most types of data flow graphs. Conditional selection of already existing values is explicitly specified by the T/F (true/false) selection node as shown in Figure 4.2-1.

4.2.2. Merging data flow graphs

In Section 4.1.1, we have seen an example of pipelining a sequence of identical computation tasks, where the data flow of each task is fixed without any conditional operations which need to be performed only on certain conditions. However, in most complex digital systems, there are many different types of computation tasks. For example, for a microprogrammed CPU, there may be as many different types of computation tasks as the number of microinstructions. Also, even a single complex computation task may involve conditional execution of certain operations and/or conditional selection of input or output values. There are three different approaches to the synthesis of a system with many different computation tasks:

1. Use multiple parallel pipelines each of which executes a set of computation tasks.

2. Use a single dynamically reconfigurable pipeline and reconfigure the pipeline according to the computation task to execute.

3. Use a single pipeline with multiple execution paths which can be selected conditionally, and select an execution path according to the computation task to execute.

The solution technique for the third approach is essential to the first two approaches. The first approach can be implemented by iterating the solution technique for the third. The second approach also requires the synthesis and analysis techniques of the solution technique for the third. Also, in the third approach, conditional selection of input or output values of operations can be modeled using the same constructs used for the conditional operations, which we will discuss in the following sections. For this reason, we discuss only the third approach in this thesis.

However, note that **a data flow graph does not have to be a single connected graph**. A number of disconnected graphs automatically imply

multiple, parallel data flow. With parallel pipelines we assume all the parallel data flows are synchronized and controlled by a single centralized controller.

4.2.3. Conditional execution paths

If we want to design a single pipeline which will execute more than one type of computation task, we need to represent the data flow of all the computation tasks as a single data flow graph. In such a data flow graph, the actual execution path[18] of each computation task can be represented and selected by conditional branches, depending on the type of computation task to be executed. There are many constructs which can represent conditional execution paths in a data flow graph, e.g., the OR-FORK and OR-JOIN [Estrin 78], the *tokens* [Dennis 74], and the SELECT [Snow 78, Davis 82]. As long as conditional execution paths can be represented in data flow graphs, any of these constructs can be used. However, in this thesis, we will use the *distribution-join* pair, which we will discuss in detail here, in order to maintain compatibility with other parts of the USC ADAM system [Granacki 85].

Explicit conditional execution paths: A distribution node and a join node are always used as a pair. Whenever an execution path is to be selected by some condition, a distribution node must be used to split the values to every possible execution path. Whenever the execution path is no longer dependent on the condition, a join node must be used to indicate the termination of conditional executions. A join node collects all the arcs which carry the values that might be changed or produced by conditional execution paths. Only the selected values are passed through a join node.

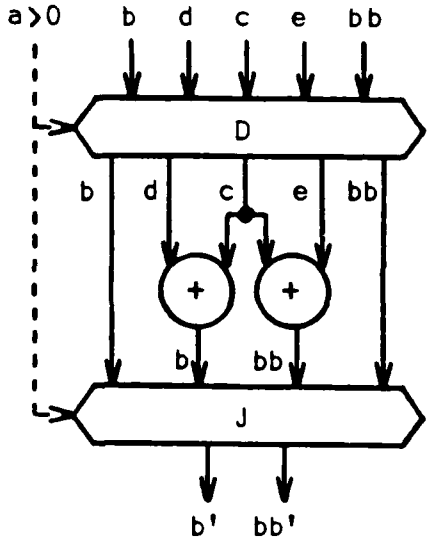
Figure 4.2-3 shows two example usages of *distribution-join* pairs. The D nodes and J nodes in this figure correspond to the distributions and joins,

---

[18] In general, each task will traverse a subgraph of the data flow graph.

```
IF a > 0
    then b  := c + d;
    else bb := c + e;
```

```
IF a > 0
    then b := c + d;
    else b := c + e;
```



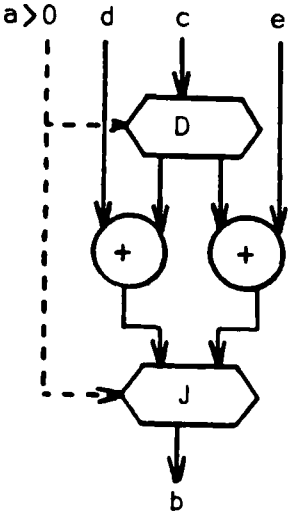


Figure 4.2-3: data flow graphs with conditional execution paths.

respectively. In both (a) and (b) of Figure 4.2-3, the IF statements are embedded in a pair of distribution and join nodes. Depending on the condition, $a > 0$, only certain output values of the D node become available, and only the subgraph reachable from the output arcs of the D node is traversed or executed.

As shown in Figure 4.2-3-(b), as long as conditional execution paths can be identified, not all the values read during conditional paths need to pass through the distribution node.

Figure 4.2-4 shows an example of nested conditional execution paths. As shown in the figure, a conditional execution path or operation can be detected by traversing the subgraph between a distribution node and its matching join

node. In Section 4.5, we will discuss a node coloring technique which shows whether an operation is conditionally performed, and if it is, which nodes it is mutually exclusive with.

Definition 4.2.1: For any two operations in a data flow graph which are executed on some condition, if the condition that selects one operation always falsifies the condition selecting the other, and vice versa, then the two operations are called **mutually exclusive** to each other.

```
IF c = 0
    then
        x := a + b;
    else
        x := a - b
        IF x < 0
            then
                x := x + 1;
```

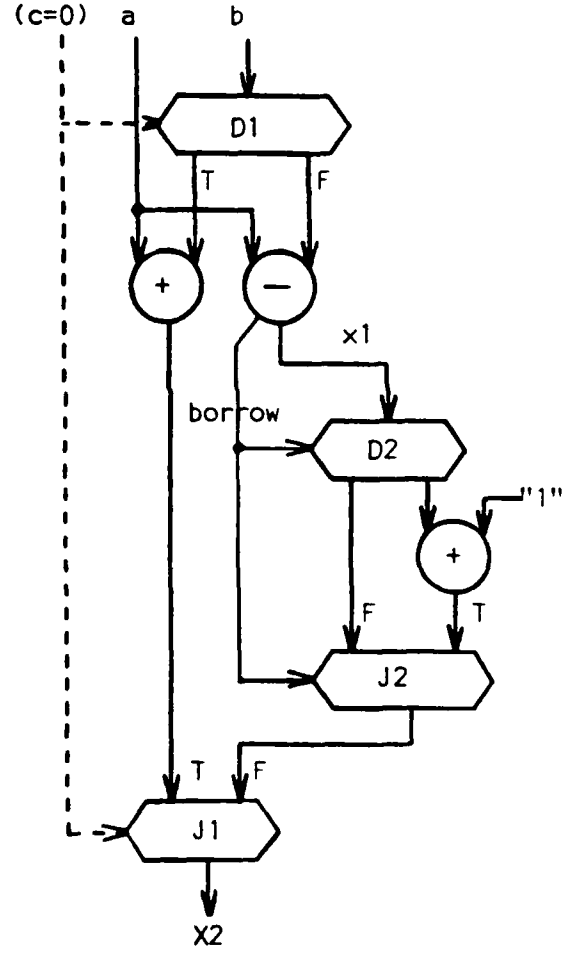


Figure 4.2-4: An example of nested distribute-join pairs.

4.2.4. Conditional value selection

The straight forward usage of a *select* node is to simply select a value from two or more values, as shown in Figure 4.2-1.

A more complicated usage of the select nodes is simplification or elimination of the distribute-join pairs. Figure 4.2-5 shows two examples of such transformations.

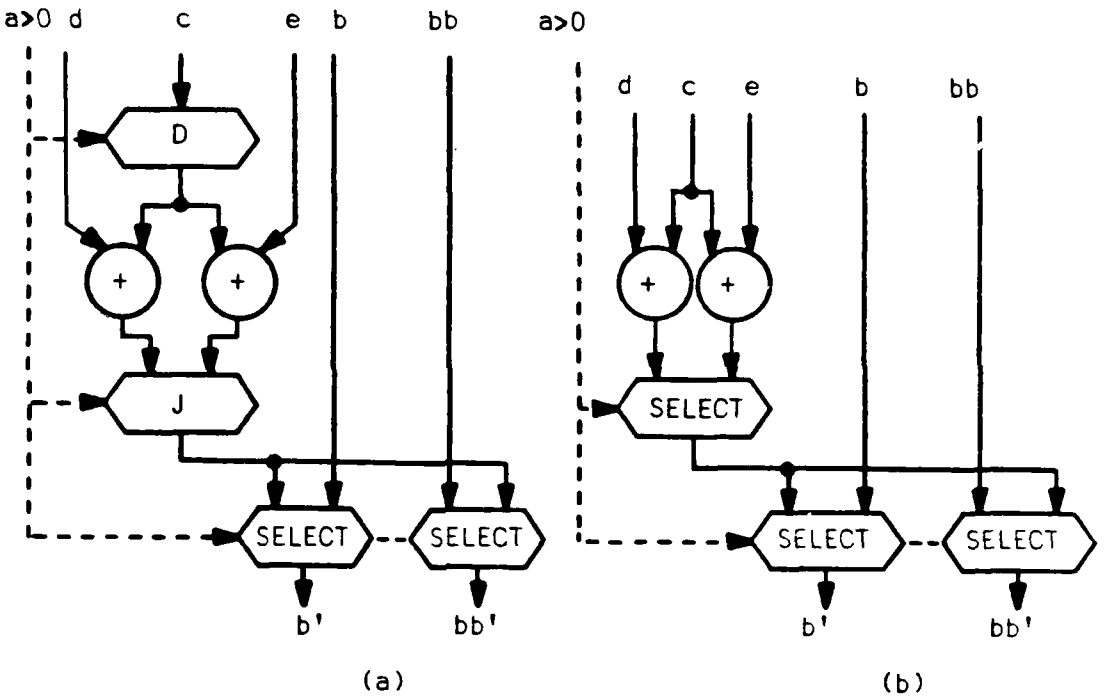


Figure 4.2-5: Transformations of the data flow graph of Figure 4.2-3: (a) simplification and (b) elimination of the distribute-join pair.

In (a), the distribute-join pair of Figure 4.2-3 is simplified by separating out the values which can be simply selected. As shown in Figure 4.2-5-(a), the data flow graph shows the conditional execution paths explicitly. However, in (b), the distribute-join node pair is completely eliminated, and the conditional execution paths become implicit. In this thesis, we will treat such implicit

conditional execution paths as parallel execution paths. In other words, by replacing the distribute-join constructs with select nodes, *the user can force parallel separate evaluations of mutually exclusive operations.* This type of transformation can speed up the execution in cases when the condition selecting which operations to perform is not available before the input values of the conditional operations are. For the data flow graph in Figure 4.2-5-(a), the condition, a $>$ 0 must be available before one of the conditional additions start. However, in Figure 4.2-5-(b), the additions can start whenever their inputs are ready regardless of the availability of the condition. Then, the appropriate outputs can be selected when the condition is available.

On the other hand, there is also a way of forcing module sharing using only select nodes. Figure 4.2-6 shows a further transformation of the data flow graph of Figure 4.2-5. In this case, *conditional execution paths are completely replaced by conditional selection of values and module sharing is explicitly forced.*

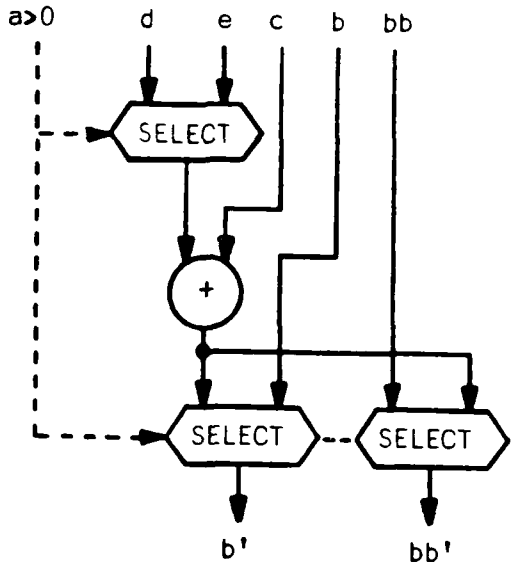


Figure 4.2-6: Transformation of the graph of Figure 4.2-5-(b).

4.2.5. Insertion of no-operation (NOP) nodes

NOP (no operation) nodes are dummy nodes which do not perform any real operation at all. There are several advantages to introducing NOP nodes:

1. It makes the data flow graph more readable to human designers as well as easily manageable for automatic procedures.

2. It allows insertion of arbitrary time delays.

An example of the first case is illustrated in Figure 4.2-7 with two equivalent data flow graphs.

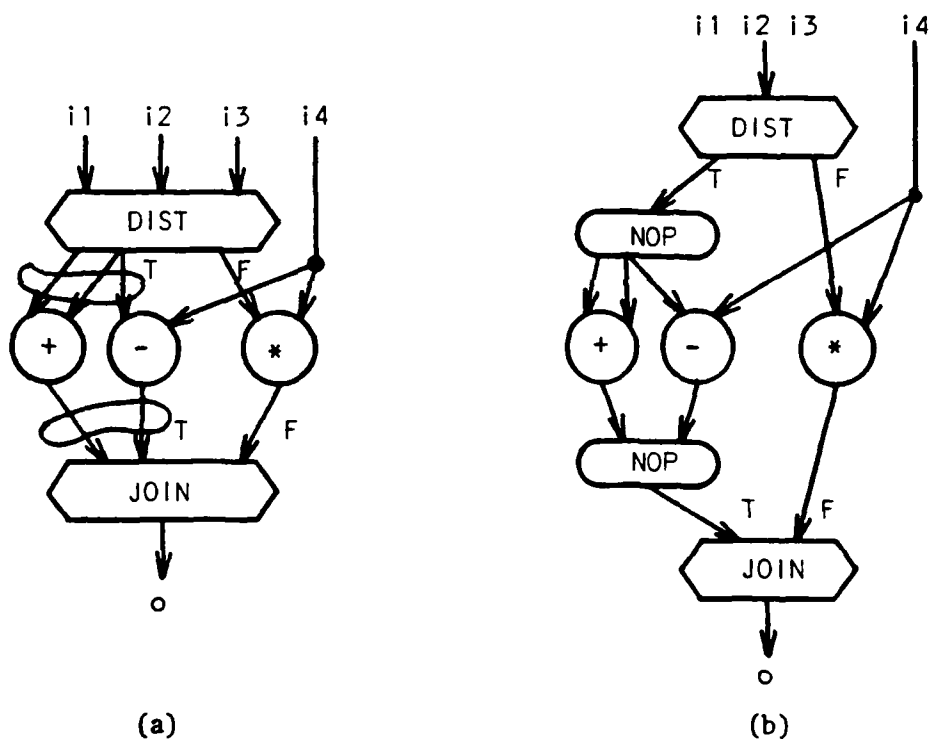


Figure 4.2-7: An example usage of NOP nodes.

In both (a) and (b) of Figure 4.2-7, there are two conditional execution paths, one with a + and a - operation in parallel and the other with a * operation. In case (a), for both a human designer and an automatic procedure, there must be some way of specifying which arcs coming out of distribute

nodes and going into join nodes are parallel and which are conditional. However, in the case of (b), NOP nodes are used to split and concatenate values, and make it easy to distinguish parallel and conditional operations and values without any special notation. For this reason, we will force using NOP nodes for parallel execution paths inside conditional execution paths.

A simple example of the second advantage is as follows. In register-transfer level design, transmission delays due to long interconnection wires and capacitance of various parts of VLSI layers and interconnections are usually ignored. Also, the delay times of functional modules may vary depending on the various conditions of the VLSI diffusion and fabrication process. The variances of timing values due to these factors can be compensated by inserting NOP nodes with delay times without actually modifying the timing values of the hardware modules case by case. Also, inversely, arbitrary timing delays for synchronization purposes can be specified.

4.2.6. Loop unrolling

No loops are allowed in the input data flow graph for pipeline synthesis except that the whole data flow graph itself represents an implicit outer loop. Even the unrolled loop of the Design Data Structure, which we have discussed in Section 4.2.1, is not allowed. Any loop must be completely unrolled so that each operation in the data flow graph needs to be performed only once for an execution of the data flow. This is necessary in order to make the data flow graph purely acyclic. In general, acyclic graphs are easier to analyze than cyclic graphs.

Loops with determinate and finite iteration counts can be unrolled easily by duplicating the loop as many times as the iteration count. Loops with conditional or indeterminate iteration counts are hard to implement in a pipeline design style. Since multiple tasks are to be executed simultaneously on

a pipeline, indeterminate loops make task scheduling (resource sharing and initiation timing between tasks) difficult.

There are two practical solutions to this problem. Suppose that we have a number of computation tasks to be executed on a pipeline and each task forms a subgraph of the data flow graph. Suppose that a subgraph, say G_i, has a loop with indeterminate iteration count. The first solution is to separate the loop and turn it into a new subgraph or a task. In other words, the task of G_i is decomposed into two or more new tasks consisting of a task for the loop and one or more tasks for the rest of the original task without the loop. For example, let I be an instruction counting the maximum number of inputs, the sum of which does not exceed some constant:

```
I ::=
   begin
           a := 1000;
           i := 1;
       loop
           a := a - x[i];
           i := i + 1;
           if (a > 0) GOTO loop
           i := i - 1;
   end
```

Instruction I can be decomposed into instructions J1, J2, J3, as follows (a conditional branch instruction is also needed after J2):

```
J1 ::=                J2 ::=                J3 ::=
    begin                 begin                 begin
       a := 1000;            a := a - x[i];        i := i - 1;
       i := 1;               i := i + 1;           end
    end                   end
```

Any program with instruction I can be rewritten using these decomposed

instructions together with a conditional branch instruction. An example is shown below. In this example, program A with an I instruction is rewritten to program B without the I instruction.

```
program A                    program B
    .                            .
    .                            .
    I;                           J1;
    .                       loop: J2;
    .                             IF (a>0) JUMP to loop;
    .                             J3;
    .                             .
```

In program B, there is no single instruction with any loop with indeterminate iteration count in it. In other words, the loop inside instruction I is removed by having a loop in the program. Therefore, in the data flow graph representing all the instructions, there will be no loops with indeterminate iteration count.

The second solution is to make exceptions. A simple solution is not initiating the next task until the execution of such a task with an indeterminate loop completes. This can be implemented by treating such a task the same as a branch task, with the next task to be executed determined on its completion.

4.3. Scheduling and Resource Allocation

Scheduling assigns operations to time steps within a maximum stage time limit. The stage time limit may be either forced by some external clock source or desired for speed optimization. Resource allocation assigns resources to each time step within the cost constraint. How many of each type of modules to use and how to share the modules between subtasks are all resource allocation problems.

4.3.1. Relation between the scheduling and resource allocation tasks

The results of scheduling affect resource allocation and vice versa. For example, a schedule forces resource allocation for each subtask, although it does not fix exactly how to share resources between subtasks. In other words, each operation assigned to the same time slot must be assigned to a different module, even though the same module can be reused for other operations in other time steps. Inversely, scheduling of each stage is affected by the the total number of available modules.

Even with the same total cost, the performance of the design may vary depending on the interaction between these two tasks. For example, even with a fast (short) schedule, an unbalanced resource allocation may force use of a module more often than others and results in a long initiation interval due to the resource conflict between consecutive tasks for that module. For this reason, these two tasks must be performed together when an optimal design is desired with either a total cost limit or a minimum performance requirement.

4.3.2. Basic requirements of scheduling and resource allocation

We briefly summarize the requirements of scheduling and resource allocation, which we will discuss in detail in the following sections.

Both scheduling and resource allocation must be performed in such a way that the result preserves the behavior of the original data flow as well as meets the design constraints.

Data precedence: The result of partitioning a data flow into time steps must retain the data precedence between operations in the original data flow graph. An operation can start execution only after all its input values are available. Thus, a node in a data flow graph must be assigned at the earliest time to the same time step as any of its predecessors in order to preserve data precedence.

Cost constraint: Another constraint on scheduling and resource allocation is the maximum available cost budget or the number of available modules. There are two types of resource constraints on scheduling and resource allocation in order not to have any resource conflicts.

1. The resource requirement of each subtask assigned to a clock cycle must not exceed the total available set of resources.

2. The sum of the resource requirements of the subtasks of different tasks which are to be executed during the same clock cycle must not exceed the total available resources.

Stage time limit: When a pipeline is being designed as a part of a larger system, the pipeline under design might be forced to use an existing external clock source whose timing is already fixed. Even when the pipeline is designed separately, the designer might want to optimize the speed and limit the slowest clock rate. In other words, the schedule must be done in such a way that each

subtask can complete within a clock cycle, including the stage latch delays as we have discussed in Section 2.3.3.

Minimum required performance: When the minimum required performance is given as a design constraint, the scheduling and resource allocation should be done in such a way that the effective execution speed of the pipeline considering resynchronization overhead meets the performance constraint.

4.3.3. Performance estimation

Performance estimation of pipelines can be done using the performance estimation methods we have developed in Chapter 2. The only differences are that the initiation interval of tasks is a clock cycle times the latency instead of a clock cycle and the execution time span of a task is P times a clock cycle instead of D_S, where P is the *pipe cycle* or the number of clock cycles needed for a task to complete.

Assuming that each resynchronization event delays the initiation of the next task until the first initiation clock cycle after the completion of the current task, we can compute the execution time of a sequence of tasks using Theorem 4.3.1.

Theorem 4.3.1: On a pipeline with pipe-cycle P and a fixed latency l, the execution time T of a sequence of n tasks, where n_b of them cause resynchronization, is

$$T = \{(n - n_b - 1) + \lceil \frac{P}{l} \rceil \cdot (n_b + 1)\} \cdot l \cdot t_{cy} \qquad \square$$

The proof of this theorem is the same as that of Theorem 2.4.3 with t_{cy} substituted by $l \cdot t_{cy}$ and D_s by $P \cdot t_{cy}$.

Corollary 4.3.2: The average initiation interval for an infinitely long sequence of tasks on a pipeline is

$\{1 + (\lceil \frac{P}{l} \rceil - 1)\cdot\rho\}\cdot l\cdot t_{cy}$ where ρ is the average resynchronization rate.

Proof: From Theorem 4.3.1, the average execution time of a task, T_{task}, can be computed by dividing the total execution time, T, by the number of tasks executed, n.

$$T_{task} = \frac{T}{n} = \{(1 - \frac{n_b}{n} - \frac{1}{n}) + \lceil \frac{P}{l} \rceil \cdot (\frac{n_b}{n} + \frac{1}{n})\}\cdot l\cdot t_{cy}$$

$$\lim_{n \to \infty} T_{task} = \{(1 - \frac{n_b}{n}) + \lceil \frac{P}{l} \rceil \cdot \frac{n_b}{n}\}\cdot l\cdot t_{cy}$$

By replacing $\frac{n_b}{n}$ with ρ, we get

$$\lim_{n \to \infty} T_{task} = \{(1 - \rho) + \lceil \frac{P}{l} \rceil \cdot \rho\}\cdot l\cdot t_{cy}$$

$$= \{1 + (\lceil \frac{P}{l} \rceil - 1)\cdot\rho\}\cdot l\cdot t_{cy}$$

□

4.3.4. Cost constrained scheduling and resource allocation

Either with execution overlap or without, the number of modules required for any task or subtask executed during a clock cycle cannot exceed the total number of available modules. For pipelining, all the modules allocated to a subtask are activated during the clock cycle executing that subtask. Accordingly, the first restriction on scheduling and resource allocation can be stated as follows:

Let R_i be the set of resources to be allocated to a subtask, F_i, and R_{total} be the total set of available modules. Then the resource allocation must satisfy

$$R_i \subseteq R_{total} \text{ and } \bigcup_i R_i \subseteq R_{total}, \forall i. \qquad (4.3.1)$$

Since subtasks of different tasks are to be executed overlapped, Equation (4.3.1) is not a sufficient condition for no resource conflicts. When multiple subtasks are to be executed at the same time, the sets of resources the subtasks use must be disjoint in order not to have any resource conflicts.

Theorem 4.3.3: For a schedule of a task partitioned into n subtasks, F_1 through F_n, a fixed latency of l can be used if and only if the resource allocation satisfies the following condition:

$$\text{For every i, } 1 \leq i \leq l, \bigcap_k R_{i+kl} = \{\emptyset\}, \forall k, 0 \leq k \leq \lfloor \frac{n-1}{l} \rfloor$$

Proof: If a new task is initiated at every initiation interval, for every i, $1 \leq i \leq l$, the subtasks, $\{F_{i+kl}\}$, $0 \leq k \leq \lfloor \frac{n-1}{l} \rfloor$, of k consecutive tasks are executed during the same clock cycle. If any initiations of tasks are skipped, a subset of the subtasks in this set are executed during the same clock cycle. Therefore, the condition in Theorem 4.3.3 is a necessary and sufficient condition for no resource conflict. □

Corollary 4.3.4: For a schedule of a task partitioned into n subtasks, F_1 through F_n, if a fixed latency of l is used, for any i, $1 \leq i \leq l$, any two subtasks which are not in the same set, $\{F_{i+kl}\}$, $0 \leq k \leq \lfloor \frac{n-1}{l} \rfloor$, can share the same module without any resource conflicts. □

In the scheduling of Figure 4.1-4, there are two sets of subtasks which are executed at the same time: $\mathcal{F}_1 = \{F_1, F_3, F_5\}$ and $\mathcal{F}_2 = \{F_2, F_4\}$. Any two subtasks in the same set, $\mathcal{F}_i$, cannot share the same module, since they are executed during the same clock cycle. Inversely, any two subtasks that are not in the same set can share the same module since they never are executed in parallel. In the case when the fixed latency of a pipeline is 1 as shown in Figure 4.1-2-(a), any two subtasks can be executed in parallel, and therefore, there cannot be any resource sharing between any two subtasks.

Figure 4.3-1 shows examples of cost constrained scheduling and resource allocation. (a) shows a data flow graph with 5 addition operations (+1 through +5) and 2 multiply operations (*1 and *2). Assuming that there are two adders, A1 and A2, and a multiplier, M1, (b) and (c) show two possible schedules and resource allocations. In case (b), any fixed latency from 1 to 4 will cause resource conflicts. Fixed latencies 1, 3, and 4 will cause resource conflict between any two consecutive initiations of tasks. In the case when fixed latency 2 is used, there will be resource conflicts between every other initiations of tasks, i.e., between F5 of a task and F1 two tasks later, for the adder, A1. In case (c), the length of pipe cycle is increased by one over (b). This schedule and resource allocation can be implemented with fixed latency 3 as shown in (d).

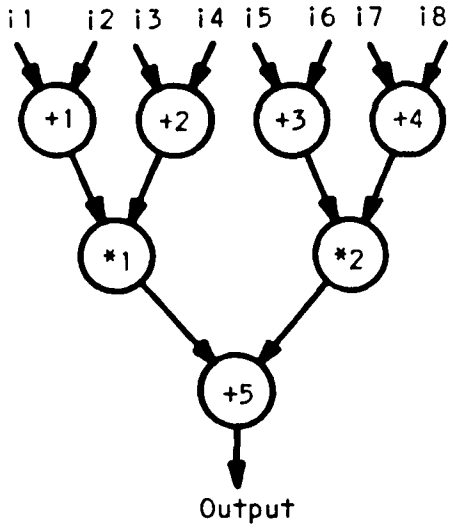
4.3.5. Speed constrained scheduling and resource allocation

As we discussed in Section 4.3.3, the speed of a pipeline is determined by the fixed latency, the clock cycle time, the length of the pipe cycle, and the rate of resynchronization. Among them, the fixed latency and the clock cycle time determine the maximum possible initiation rate.

In the case when the minimum required performance is given, we can compute the possible combinations of latencies and stage times[19] which can meet the minimum performance requirement. Among all the possible combinations of latencies and stage times, we want to choose one with minimum resource requirements. Among all the resource requirements, the cost for the operators is the major contributor to the total cost. In addition, the number of multiplexers, latches, and wiring space can be estimated from the number of operators [Sastry 83, Kurdahi 85].

---

[19] Note that there are only $O[n^2]$ possible stage times as we have discussed in Chapter 2.



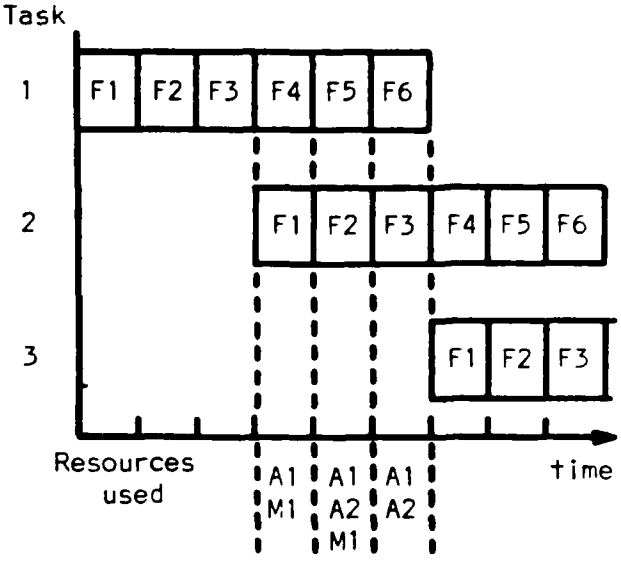
(a) A data flow graph

Subtask	Operation	Resource
F1	+1, +2	A1, A2
F2	+3, +4	A1, A2
F3	*1	M1
F4	*2	M1
F5	+5	A1

(b) A schedule of five cycles and resource allocation

Subtask	Operation	Resource
F1	+1	A1
F2	+2, +3	A1, A2
F3	+4	A2
F4	*1	M1
F5	*2	M1
F6	+5	A1

(c) A schedule of six cycles and resource allocation



(d) Execution timing of schedule (c) with fixed latency 3.

Figure 4.3-1: An example of cost constrained scheduling and resource allocation.

For these reasons, we want to compute the minimum required number of operators with a certain fixed latency. This can be done by taking a reverse approach to the cost constrained scheduling and resource allocation case. In other words, we can force the scheduling and resource allocation in such a way that the number of modules can be minimized while satisfying Theorem 4.3.3.

Theorem 4.3.5: Suppose that there are at maximum N_i operation nodes in a data flow graph which **must be performed**[20] by type-i modules during an execution of a task. Then the minimum **necessary and sufficient** number of type-i modules for a schedule and resource allocation with a fixed latency l is $\lceil \frac{N_i}{l} \rceil$.

Proof: As shown by Corollary 4.3.4, there are l sets of subtasks, $\mathcal{F}_1$ through $\mathcal{F}_l$, where

$$\mathcal{F}_i = \{F_{i+kl}\}, \quad 0 \le k \le \lfloor \frac{n-1}{l} \rfloor ,$$

between any two sets of which module sharing is possible without resource conflicts. When N_i operations are distributed into l such sets of subtasks, the smallest possible maximum number of operations in a set is $\lceil \frac{N_i}{l} \rceil$ (sufficient condition). Since all the modules in each such set of subtasks must be active during the same cycle, $\lceil \frac{N_i}{l} \rceil$ is the minimum necessary number of type-i modules (necessary condition). □

Corollary 4.3.6: When a fixed latency l is used, as long as a module is not used in any two clock cycles which are l cycles apart, there is no resource conflict. Accordingly, *a module can be used at maximum in l subtasks whose indices modulo-l are distinct and (0, 1, 2, .. , l-1).*

Suppose that we want to design a pipeline with fixed latency 2 for the

---

[20] Any number of mutually exclusive operations which use the same module are counted as one.

data flow graph of Figure 4.3-1-(a). According to Theorem 4.3.5, we need at least $\lceil \frac{5}{2} \rceil = 3$ adders and $\lceil \frac{2}{2} \rceil = 1$ multiplier.

The schedule and resource allocation for this design with three adders, A1, A2, and A3, and a multiplier, M1, is shown in Figure 4.3-2 with an execution timing diagram.

Subtask	Operation	Resource
F1	+1,+2,+3	A1,A2,A3
F2	+4, *1	A1, M1
F3	*2	M1
F4	+5	A2

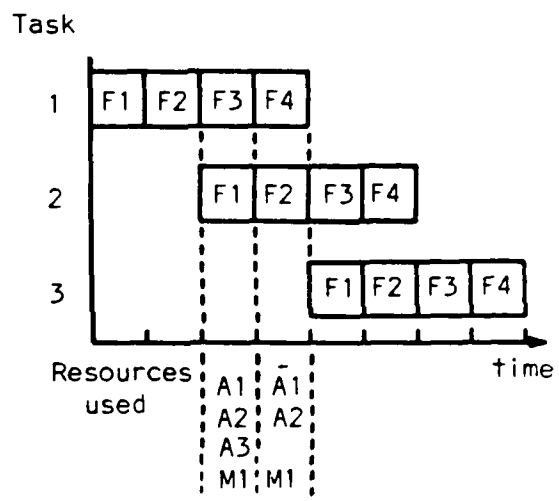
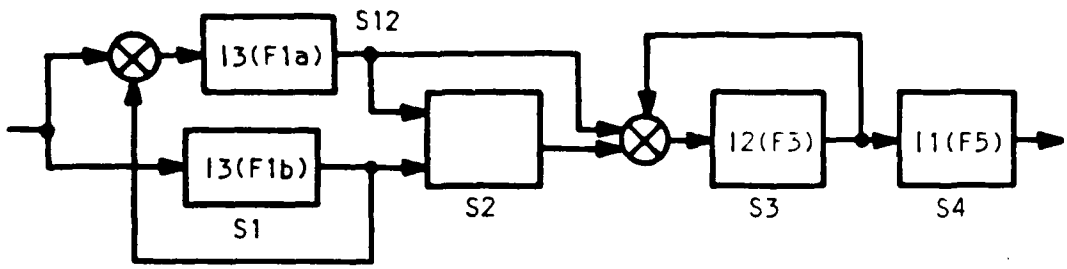


Figure 4.3-2: A scheduling and resource allocation with fixed latency 2 for the data flow graph of Figure 4.3-1.

4.3.6. Patterns of resource allocation

In Section 4.1.1, we showed examples of two different pipeline implementations with different costs and speeds. Another cheap design of a pipeline for the partitioned data flow graph of Figure 4.1-1 is shown in Figure 4.3-3.



F1=F1a// F1b

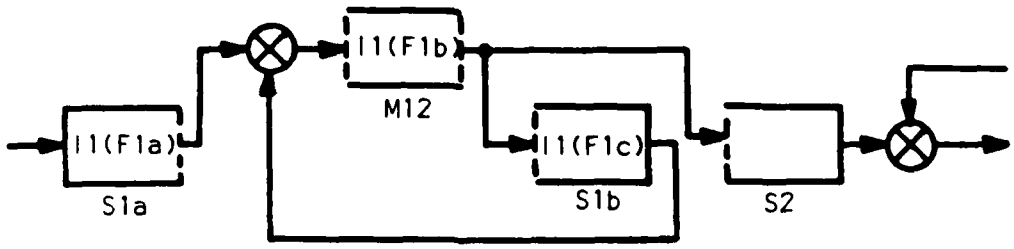
Note: During the next clock cycle, S2 and S12 will execute I3(F2)

Figure 4.3-3: Another cheap pipeline design.

In this design, only one stage, S12, is reused. Stages S1 and S12 in parallel perform F1, and stages S2 and S12 in parallel perform F2. In both cases, for any stage, an entire stage is used or reused to perform a part of or an entire subtask. The timing diagram for this pipeline is the same as that of Figure 4.1-4.

Figures 4.1-3 and 4.3-3 show the only two patterns of resource sharing used in the previous pipeline scheduling work discussed in Section 1.3. However, in this thesis, we generalize the concept of resource sharing to non-pipelined design. Figure 4.3-4 shows an example of such generalized resource allocation with resource sharing.

For the pipeline of Figure 4.3-4, subtasks F1 and F2 are further decomposed into sequences (F1a - f1b - F1c) and (F2a - F2b), respectively. The shared module, M12, is shared for F1b and F2a, and is not a part of either S1 or S2. Suppose that the rest of the pipeline stages, S3 and S4, are the same as in the pipeline of Figure 4.3-3. Then this pipeline can also implement the schedule of Figure 4.1-4. An example of this type of resource sharing is shown in Figure 4.3-5, where module A is used in three different stages.



F1=F1a·F1b·F1c

Note: During the next clock cycle, S2 and M12 will execute I1(F2)

Figure 4.3-4: A more generalized form of shared resource allocation.

For this type of general resource sharing pattern, the reservation table method [Davidson 71], which has been an essential tool for pipeline scheduling, is no longer valid. This is due to the fact that each row in the reservation table corresponds to a pipeline stage and no resource in a stage can be reused in any other stages. In the case of the pipeline in Figure 4.3-5-(b), module A is sometimes used alone as a single stage and sometimes used in different stages together with other modules. For this reason, we need some other representation methods for resource scheduling, which we will discuss in later sections.

In this thesis, we allow all three patterns of resource sharing. In fact, the two resource sharing patterns shown in Figure 4.1-3 and 4.3-3 are special cases of the resource sharing pattern shown in Figure 4.3-4.

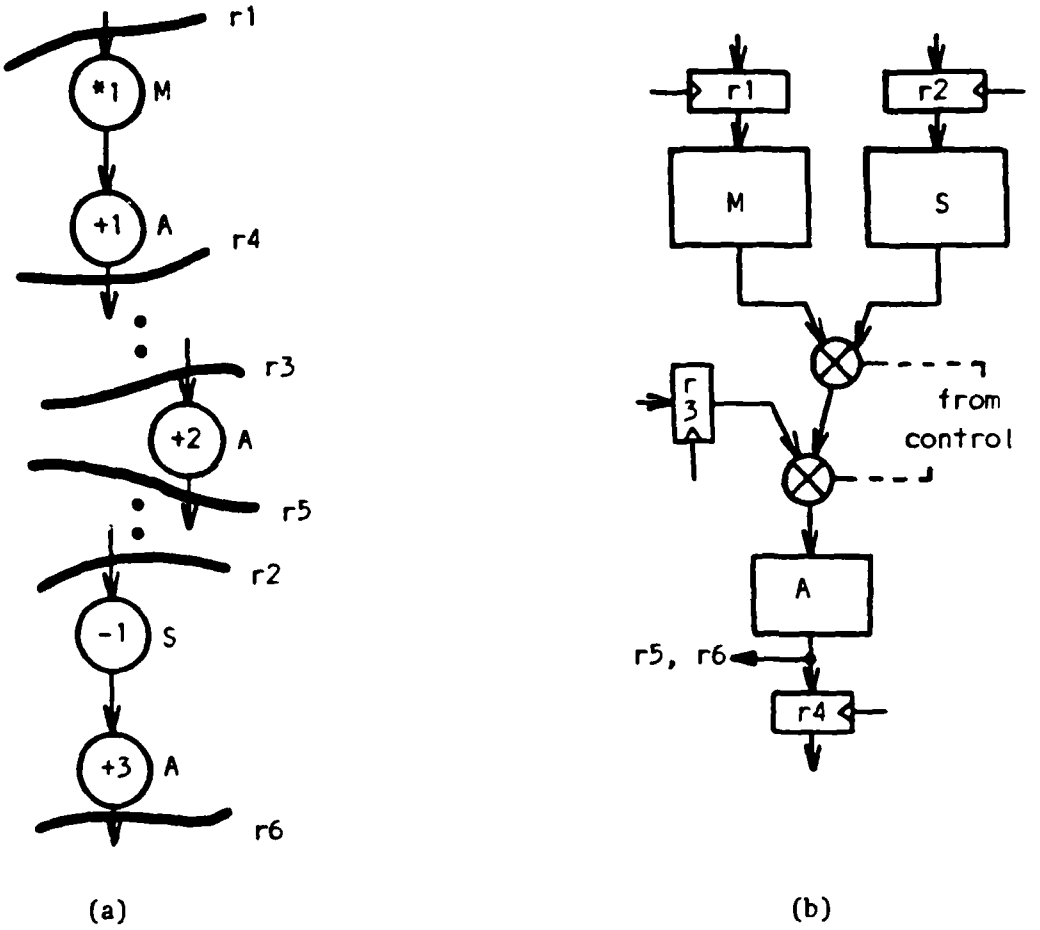


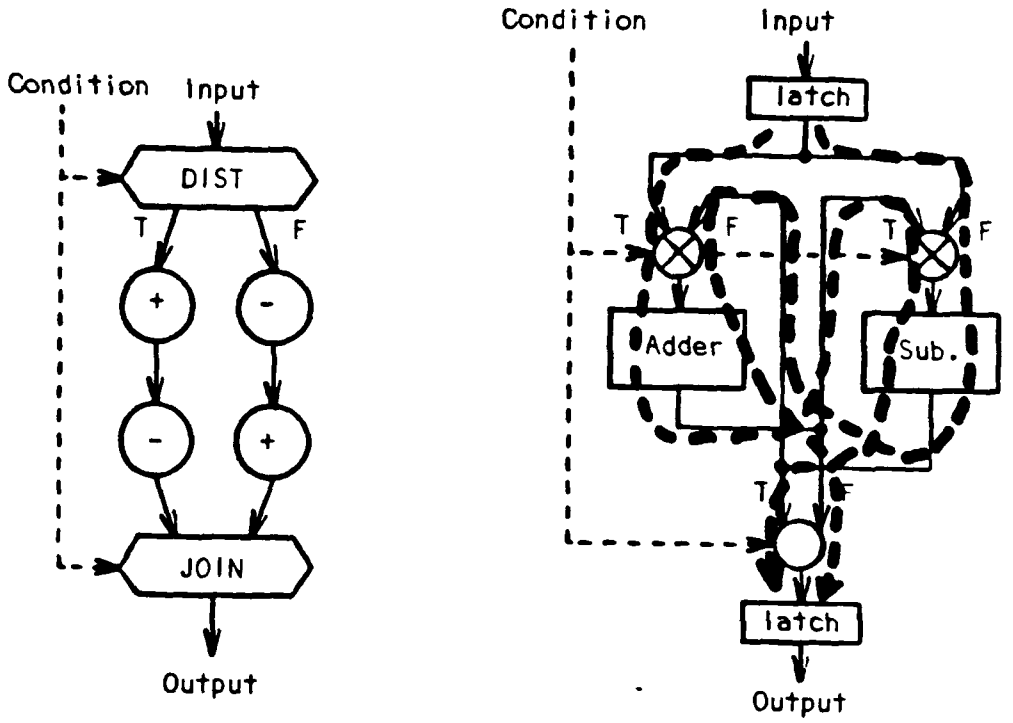
Figure 4.3-5: An example of generalized resource sharing: (a) a staged data flow graph and (b) a pipeline implementation.

4.4. Resource Sharing between Mutually Exclusive Operations

Up to this point, we have ignored conditional branches in our analysis. In this section, we discuss how the theory of scheduling and scheduling we developed in the previous sections can be applied to data flow graphs with conditional branches.

4.4.1. Conditional execution paths and stage construction

Let us first see how the actual stage construction looks when we have mutually exclusive operations of the same type. The following example will give us a clear explanation why we counted only the maximum number of the operations which are actually performed. Figure 4.4-1 shows an example of such a stage construction. The thick broken lines in Figure 4.4-1-(b) show two possible execution paths for the subtask in Figure 4.4-1-(a).



(a) A subtask with conditional execution paths

(b) Stage implementation with shared modules A and S

Figure 4.4-1: Resource sharing between mutually exclusive operations within a subtask.

In this fashion, any number of mutually exclusive operations which can be performed by the same type of module can be assigned to a single module in

a subtask. However, such sharing might increase the control and wiring complexity and the execution time due to the multiplexing requirements for the selection of one conditional execution path.

In order to compute the minimum set of operators with Theorem 4.3.5, we first need to compute the maximum possible number of operations of each type actually performed per task. We will discuss a technique for this computation after we discuss a graph coloring technique for automatic mutual exclusion testing between conditional operations in the next section.

4.4.2. Conditional and unconditional resource sharing

As we discussed before, there are two different types of resourcing sharing. They are

1. between operations across non-overlapping time steps, and
2. between mutually exclusive operations.

In order to clarify the difference between these two types of resource sharing, we define the following two terms:

Definition 4.4.1: Unconditional resource sharing shares resources across non-overlapping time steps and **conditional resource sharing** shares resources between mutually exclusive operations.

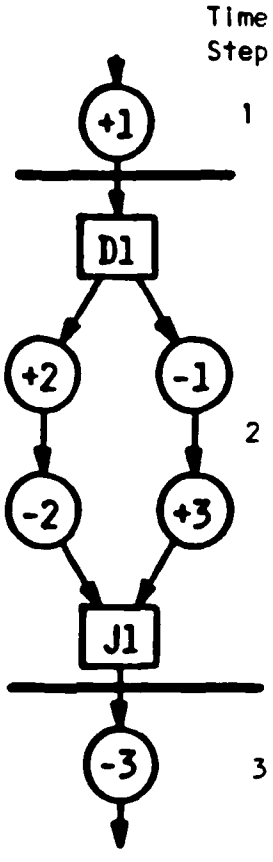
As we have shown with Corollary 4.3.6, the maximum unconditional resource sharing is determined by the number of modules and the chosen fixed-latency. It specifies how many times a module can be used during a computation of a task.

Conditional resource sharing does not depend on either the latency or the number of available modules. Instead, it specifies how an instance of usage of a

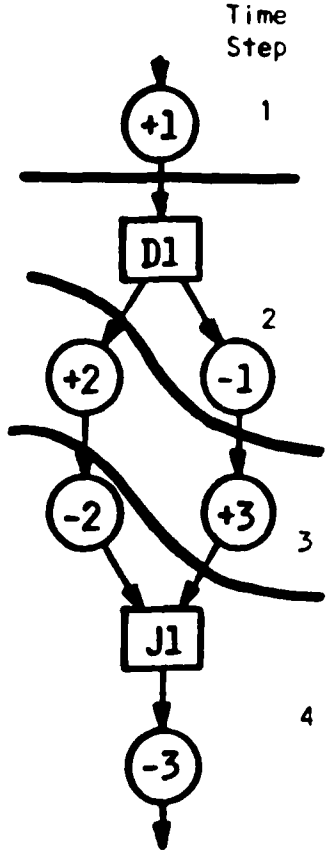
module can be shared between mutually exclusive operations. For example, suppose that a module is assigned to certain time steps according to the unconditional resource sharing rules. In a time step where such a module is assigned, the module can be used only once in that time step. However, since only one of the mutually exclusive operations is actually performed, all the mutually exclusive operations which can be performed by the module can be assigned to the module without any resource conflict. However, conditional resource sharing across time steps is unpredictable and thus not always possible. We illustrate this with an example.

In the data flow graph of Figure 4.4-2, there are one unconditional addition and subtraction each and two conditional additions and subtractions each. Thus, at most, two additions and two subtractions are performed during an execution of the data flow graph. Suppose that we have one adder and one subtractor. Then, the minimum possible latency is 2. In schedule A, the two mutually-exclusive subtractions, -1 and -2, are scheduled in the same time step. The two additions, +2 and +3 are assigned to the same time step, too. As shown in (c), with schedule A, a fixed latency 2 can be used without any resource conflict. In schedule B, two mutually-exclusive subtraction operations, -1 and -2, are scheduled in time steps 1 and 3, respectively. With this schedule, a fixed latency of 2 can cause resource conflicts. The time steps 1 and 3 overlap. Therefore, in the timing diagram of (d), if the -2 operation is selected in I1 while the operation -1 is selected in I2, there will be resource conflicts since there is only one subtractor.

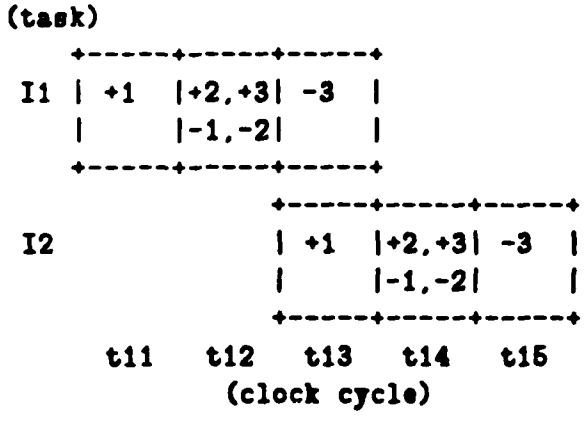
It is difficult to detect when resources can be shared under such circumstances unless exhaustive search is performed. For this reason, *conditional resource sharing (resource sharing across conditional branches) is allowed only in the same time step of a schedule.*



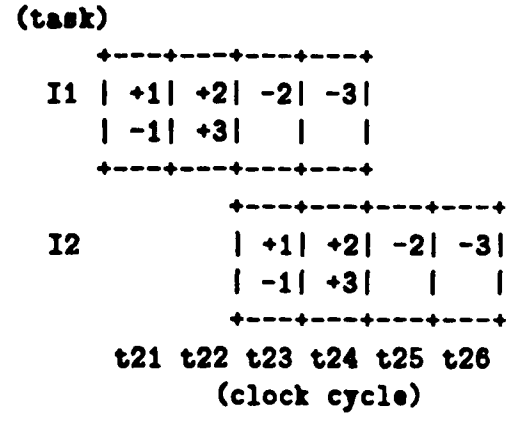
(a) Schedule A



(b) Schedule B



(c) Execution overlap of schedule-A with latency 2.



(d) Execution overlap of schedule-B with latency 2.

Figure 4.4-2: Moduling sharing between mutually exclusive operations.

4.5. Node Coloring for Automatic Testing of Mutual Exclusion

In this section, we discuss a node coloring technique for the automatic detection of mutually exclusive operations (refer to Definition 4.2.1) in a data flow graph. The node coloring algorithm we present in this section, DJ-Coloring, assigns a color code consisting of a sequence of one or more integers to each node such that testing of mutual exclusion between any two nodes can be done by simply comparing the color codes of the nodes in a constant number of steps. Note that the node coloring algorithm we present here is **not the same as the usual node coloring algorithms used in graph theory**, which assign different colors to adjacent nodes.

This preprocessing saves time during synthesis. As we will discuss in the later sections describing synthesis algorithms, whenever we partition tasks into subtasks, we need to check that there are sufficient modules to execute all the operations of the subtask. Conversely, if the task partitioning has already been done, we need to compute the minimum and maximum number of modules for the pipeline to be synthesized. Also, whenever we assign an operation to a module, we need to check whether the operation can share an already assigned module or needs to be assigned to a different module. All these design tasks require testing of mutual exclusion between every pair of operations that can share the same module. Moreover, these task partitioning and module assignment tasks are to be repeated for many possible pipeline designs. Therefore, we need a fast algorithm for mutual exclusion testing.

4.5.1. A node coloring algorithm

We first outline the rules of the node coloring. Then we outline the mutual exclusion testing procedure for any pair of nodes according to the node coloring rules. A Pascal-like description of the node coloring algorithm, which has been programmed in Franz LISP, is shown in Appendix B.

In the data flow graph of Figure 4.5-1, the circles represent real operation nodes and the squares represent distribution or join nodes. D_i and J_i are a matching pair of distribution and join nodes, forming a distribute-join block.

Definition 4.5.1: A **distribute-join block**, D_i-J_i, is a subgraph of a data flow graph consisting of all the edges and nodes that can be reached from the distribution node, D_i, without passing through the join node, J_i.

Definition 4.5.2: An **outermost distribute-join block** is a maximal distribute-join block which is not a part of any other larger distribute-join block.

In other words, an outermost distribute-join block is always entered and at least one path in it is selected during an execution of any task.

The color code of each node is parenthesized on the right side of the node. Examples cited in this section automatically refer to Figure 4.5-1, unless specified otherwise.

4.5.1.1. Node coloring rules

Each node is given a color code which is a sequence of one or more integers. The length of the color code of a node represents how many levels of distribute-join blocks the node is nested in. A single-digit color code represents that the node is not in any distribute-join block. Any node in a distribute-join block has a color code with more than one element. We first summarize the rules for node coloring as follows:

Rule 1: Unconditional operations, including outermost-level distribution and join nodes, NOP nodes, and SELECT nodes, have a single element code sequence (e.g. (0) for +2, (1) for D1, and (3) for J4.). Any operation or non-operation node that is conditionally executed has a color code of length at least 2 (e.g. (1 0) for -2, (1 1) for D3, and (1 0 1) for -5.).

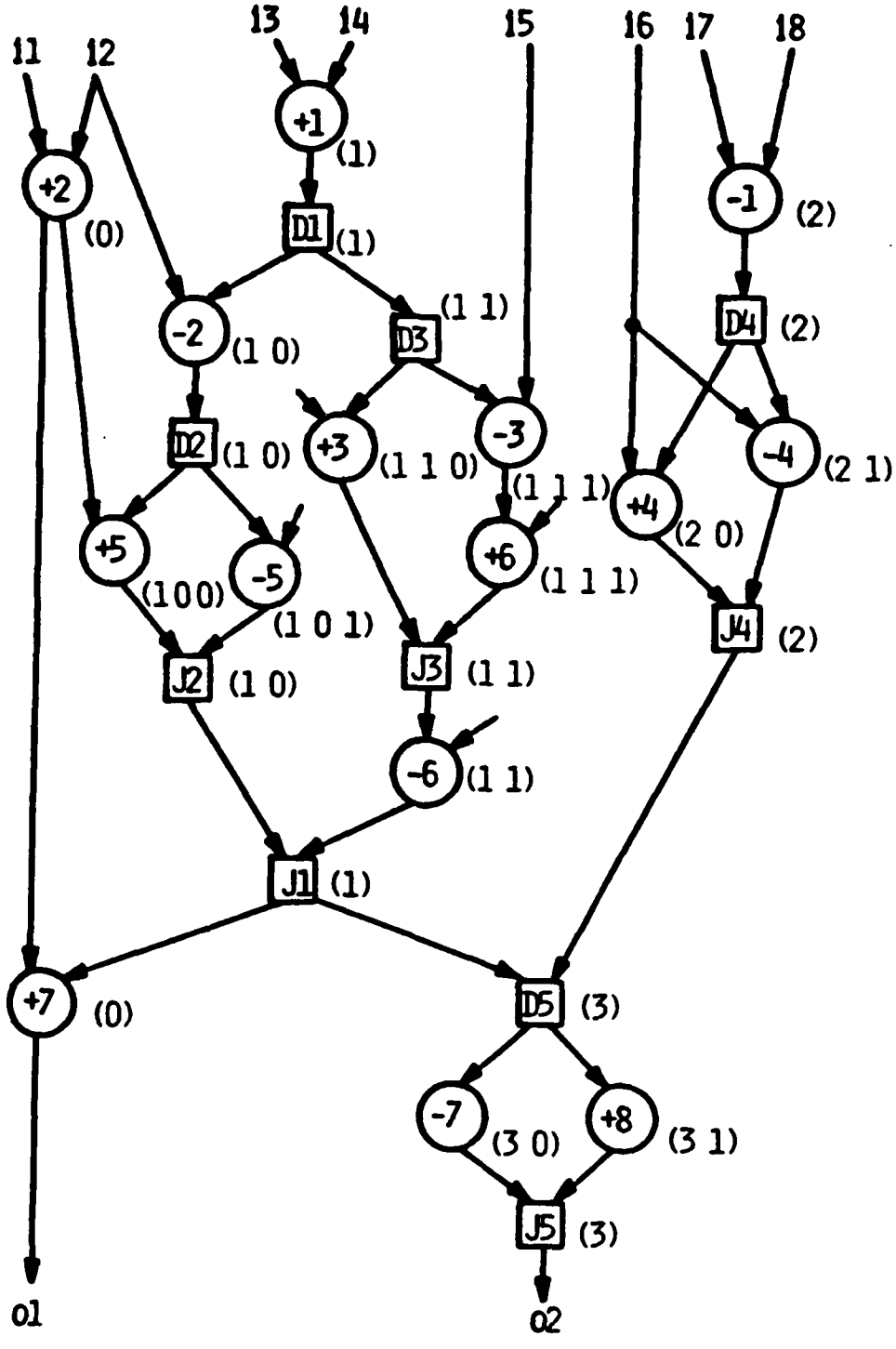


Figure 4.5-1: A node-colored data flow graph.

Rule 2: In an outermost distribute-join block, the first elements in the code sequences of all the nodes are the same (e.g. (1) for D1, (1 1) for D3, and (1 1 0) for +3).

Rule 3: The common first element of the color codes in an outermost distribute-join block is different from that of any other outermost distribute-join block (e.g. (1) for D1 and (2) for D4).

Rule 4: For any distribution node with a color code of length n, every child node except its matching join node has a color code of length n+1 where the first n elements are the same as the distribution node's and the (n+1)-th element is a unique integer among the children (e.g. (1 1 0) for +3 and (1 1 1) for -3).

Rule 5: A join node has the same color as its matching distribute node (e.g. (1) for D1 and J1, and (3) for D5 and J5).

Rule 6: For any two connected nodes, if both are either conditional or unconditional and neither of them is a distribution or a join node, they have the same color code (e.g. (0) for +2 and +7, and (1 1 1) for -3 and +6). Note that, by this rule, the +5 node inherited its color code from D2 not from +2.

4.5.1.2. A procedure for mutual exclusion testing

Before we present a procedure for mutual exclusion testing, we examine a special case which might be somewhat debatable. Mutual exclusion testing between two operations in different parallel distribute-join blocks is the case. In the data flow graph of Figure 4.5-1,

are the -4 operation in the D4-J4 block and the -3 operation in the D3-J3 (or D1-J1) block mutually exclusive?

The answer is *yes or no*. If the condition selecting the -3 operation in D3-J3 block always falsifies the condition selecting the -4 operation in D4-J4 block and vice versa, then the two operations are mutually exclusive. However, such a case is exceptional and expected to occur seldom. Even if there is such a case, such parallel conditional blocks with very tight control dependency would tend to be merged together. In this thesis, we will always treat such operations as +3 and -4 not to be mutually exclusive.

The outline of the mutual exclusion testing procedure for any two nodes in a data flow graph is as follows:

Step 1: IF either of the two nodes have a single element color code sequence, STOP and report NOT mutually exclusive. (At least a node is not conditional: Rule 1.)

Step 2: IF the first elements of the color codes are different, STOP and report NOT mutually exclusive. (The nodes are in different outermost distribute-join blocks: Rule 2.)

Step 3: IF two color codes are identical, STOP and report NOT mutually exclusive. (There is a data dependency between the nodes and if one is executed, the other must be executed too: Rule 6.)

Step 4: IF two color codes are not the same but of the same length, STOP and report **mutually exclusive**. (They have a distribution node as their closest common ancestor, thus they are mutually exclusive: Rule 4.)

Step 5: IF none of the conditions of Step1 through Step 4 is true (i.e. both color codes have more than one digit with the same first digit but their lengths are different), then delete the last element from the longer color code until both color codes have the same length (i.e., trace back from the more

deeply nested node, say n_1, to its closest ancestor that are in the same level as the other node under comparison. If the closest ancestor of n_1 is mutually exclusive with the other node, then n_1 is mutually exclusive with the other node too.)

Step 6: IF the two color codes resulting from Step 5 are identical, STOP and report NOT mutually exclusive (There is a data dependency between the nodes: Rule 6). ELSE STOP and report **mutually exclusive** (Rule 4).

All sets of mutually exclusive nodes of the same function type in the data flow graph of Figure 4.5-1 are listed below.

Node(Color)	Node(Color)	Node(Color)	Node(Color)	Node(Color)
-2 (1 0)	-2 (1 0)	-3 (1 1 1)	-5 (1 0 1)	+5 (1 0 0)
-3 (1 1 1)	-6 (1 1)	-5 (1 0 1)	-6 (1 1)	+3 (1 1 0)
				+6 (1 1 1)

4.5.2. Computation of the maximum number of actually performed operations

As mentioned before, in order to compute the minimum necessary set of modules, we need to compute the maximum number of operations which are actually performed during an execution of a task.

Lemma 4.5.3: The maximum possible number of usage of type-M modules during an execution of a data flow graph, G, is equal to the sum of

1. the number of unconditional operations assigned to type-M module and
2. for every outermost D-J (distribute-join) block, the maximum number of conditional operations using type-M modules on a directed path in it.

The proof of this lemma is obvious. The first item sums up all the

unconditional operation nodes using type-M modules, which must always be performed. The second item sums up the maximum possible number of actual usages of type-M modules in each outermost D-J block. Since no operations across D-J blocks are mutually exclusive (refer to Section 4.4.1.2), this is the minimum necessary number of usages of type-M modules.

For an example, we count the maximum possible number of subtraction operations performed during an execution of a task. There is only one unconditional subtraction node, -1. In the D1-J1 block, the maximum number of subtraction operations is 2 (either -2 and -5, or -3 and -6). Both D4-J4 and D5-J5 have only one subtraction operation. Therefore, the total count is (1 + 2 + 1 + 1) = 5. Let us also count the addition operations. +1, +2, and +7 are unconditional and the count becomes 3. +4 and +8 do not have any addition operations they are mutually exclusive to; thus, we increment the count to 5. In D1, +5, +3, and +6 are mutually exclusive to one another. Therefore we add only one to the count and the total number becomes 6.

The intuitive meaning of the maximum number of actually performed operations is the following: As we have discussed in Section 4.3.5, if we know the total number of operations actually performed using a certain type of module, we can compute the minimum necessary number of modules to achieve a certain fixed-latency (refer to Theorem 4.3.5). Conversely, if we have M_i modules of a certain type and we know the maximum number of operations which must actually be performed on those modules, we can compute the minimum possible latency.

A Pascal-like algorithm description for this counting, which has been programmed in Franz LISP, is shown in Appendix C.

4.6. Some Remarks on Cost-performance Optimization

The theorems and corollaries in the previous two sections give a good starting point for the design space exploration. However, the precise performance and cost of a pipeline design can only be estimated after register-transfer synthesis.

As we have shown with Theorem 4.3.1 and Corollary 4.3.2, the actual performance of a pipeline design is a function of not only the latency and stage time but also the length of the pipe cycle (or schedule), and the resynchronization overhead. Also, the actual cost of a pipeline design can only be estimated after all the necessary number of multiplexers and latches are determined. Reducing the number of operators does not necessarily reduce the total cost. More sharing of modules will increase the number of multiplexers. Also, the minimum number of modules computed by Theorem 4.3.5 may force scheduling to increase the length (the number of subtasks) of schedule in order to meet the given latency. Shortening the stage time for a faster clock cycle will increase the number of stage latches.

For these reasons, in order to achieve a certain required performance at minimum cost or to maximize performance at a fixed cost, we need to tune the scheduling and resource allocation by varying the number of operators, starting from the estimation we have discussed in this chapter. This requires good prediction techniques based on previous designs, which can guide the design process towards a desired direction. In the next chapter, we will discuss such techniques when we discuss synthesis algorithms and procedures.

Chapter 5
Synthesis of Pipelines

In this chapter we present some algorithms and procedures for automatic synthesis of pipelines, based on the discussions in the previous chapter. We first discuss the basic scheduling and resource allocation strategies together with several scheduling algorithms. Then we demonstrate how these scheduling algorithms can be used to design an optimal pipeline.

The main objective of the pipeline synthesis techniques in this chapter is to provide designers with a precise and efficient tool for quick design space exploration. While going through a large number of non inferior designs with various design constraints in a short time, the designer can tune the design process such that the output design meets all the constraints imposed on it and is optimal with respect to the designer's intention.

5.1. Introduction

5.1.1. Three types of polynomial-time scheduling algorithms

In this section, we will discuss three types of polynomial-time algorithms for scheduling a data flow graph. They are

1. **feasible-scheduling** - schedule a data flow graph with a fixed latency, a maximum stage-time limit (or a given clock cycle), and constraints on the total cost of the pipeline implementation,

2. **maximal-scheduling** - schedule a data flow graph as short as possible with only a maximum stage-time limit, assuming there is no cost constraint (in this case, the latency is always 1), and

3. **nonoverlap-scheduling** - schedule a data flow graph as short as possible with a maximum stage-time limit and constraints on the total number of available modules. No execution overlap is considered.

Schedule produced by the feasible-scheduling, the maximal-scheduling, and the nonoverlap-scheduling are called **feasible schedules**, a **maximal schedules**, and **nonoverlap schedules**, respectively.

The feasible-scheduling algorithm is the main pipeline scheduling algorithm with cost and performance constraints. It performs resource allocation at the same time in order to achieve maximum performance at a given cost. We will discuss this algorithm in detail in the next section. The second type of algorithm is used to estimate the absolute upper bound on performance and cost. Also, as we will discuss later in this chapter, this algorithm is used in an exhaustive scheduling routine for an optimal schedule. The third type of algorithm is used to estimate the maximum performance when no execution overlap is used. This is necessary to determine whether to use a pipeline. The last two types of algorithms will be discussed later in this section after we discuss several basic scheduling strategies.

5.1.2. Urgency measures of operations

The scheduling priority of operations used in the polynomial-time scheduling algorithms is based on an urgency measure for operations. The notion of urgency is the following: at any stage of the scheduling process, the shortest possible length of the schedule is determined by the sum of the time steps already scheduled and the critical path in the rest of the data flow graph which is not yet scheduled. Therefore, scheduling the nodes on the critical path first will always minimize the critical path length in the rest of the data flow graph, which will also minimize the length of the schedule.

Definition 5.1.1: In a data flow graph where nodes are weighted with the delays of the modules assigned to them, the **forward urgency** of a node is the longest path from an input edge of the node to an output edge of the data flow graph. The **backward urgency** of a node is the longest path from an input edge of the data flow graph to an output edge of the node.

The forward-urgency measure is used when scheduling starts from the root nodes and proceeds in the same direction as data flow, and the backward-urgency measure is used when scheduling starts from the terminal nodes and proceeds in the opposite direction to the data flow. An example of urgency measures is illustrated in Figure 5.1-1.

The difference between urgency-based scheduling and critical path scheduling is that the critical path is not fixed in the case of urgency scheduling. Instead, for any subgraph left to be scheduled, the path starting from the most urgent node is the critical path.

5.1.3. Forward scheduling and backward scheduling

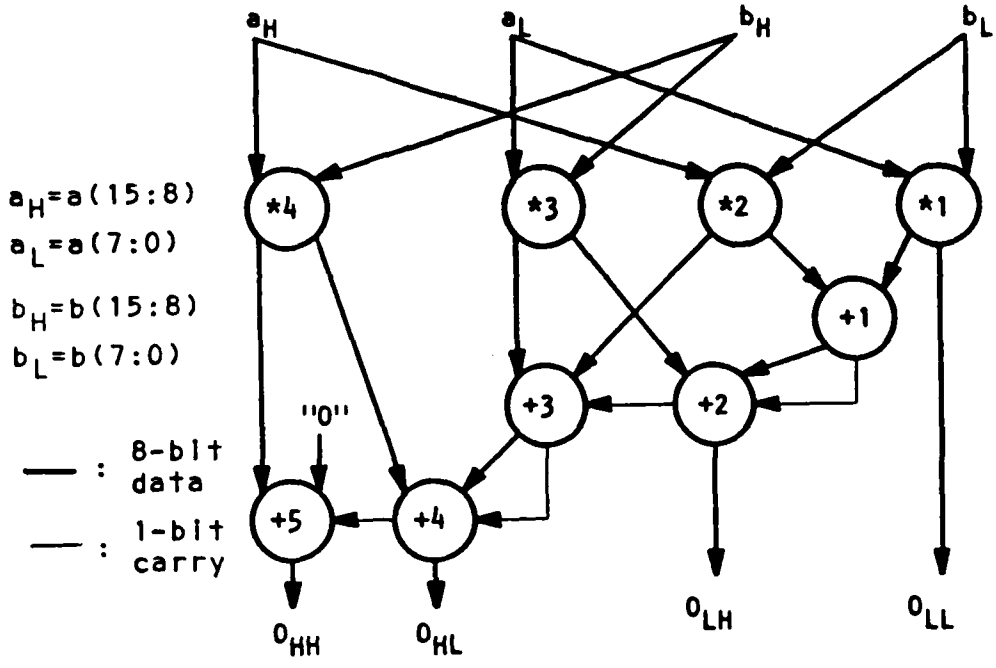
The common basic outline of all the polynomial-time scheduling algorithms we discussed before can be briefly summarized as follows:.

1. Compute urgencies (either forward or backward) of the nodes in the data flow graph.

2. For every node starting from the most urgent one down to the least urgent one, if the node does not violate any constraints, then schedule it in the current time step.

3. If there are no more nodes which can be scheduled in the current time step, then add a new time step and repeat Step 2 and 3 until there are no more nodes left to be scheduled.

There are two different approaches to each scheduling algorithm we will



Note: The multiplications *1 through *4 are assigned to multipliers with 100 nsec. delay and the additions +1 through +5 to adders with 50 nsec. delay.

node	*1	*2	*3	*4	+1	+2	+3	+4	+5
Forward Urgency	350	350	300	200	250	200	150	100	50
Backward Urgency	100	100	100	100	150	200	250	300	350

Figure 5.1-1: An example of urgency measures.

discuss, depending on which direction of urgency measures is used. One approach starts scheduling from the input side and proceeds toward the output side of the data flow graph, using the forward-urgency measures (**forward scheduling**). The other starts from the output side and proceeds toward the input side, using backward urgency measures (**backward scheduling**).

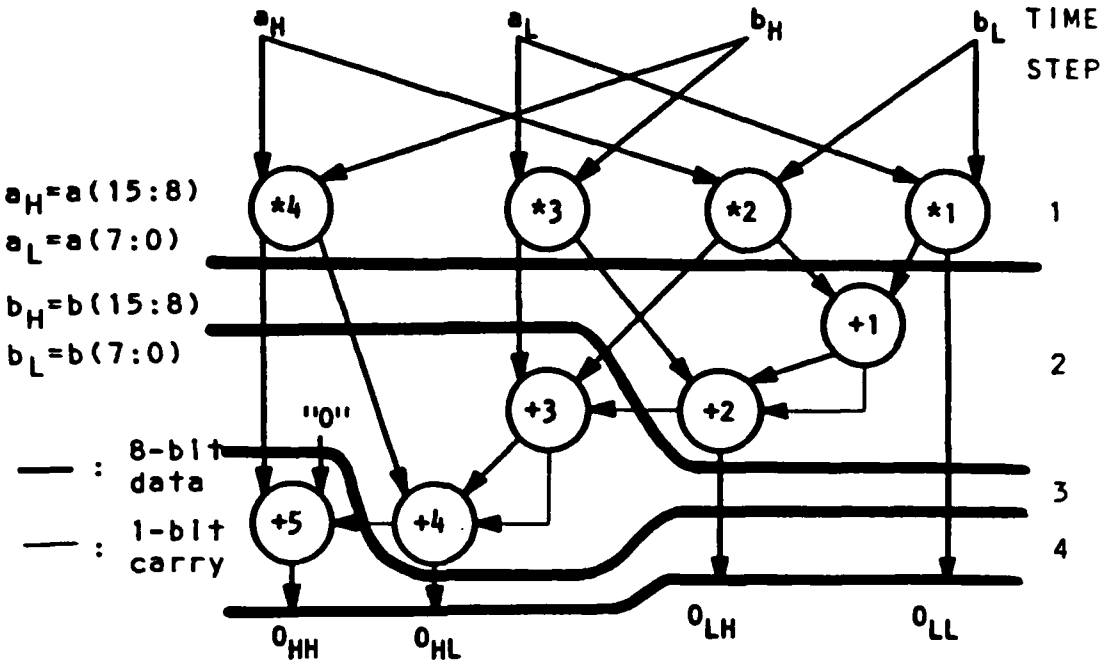
As we will see with examples, in many cases of scheduling, one approach will produce a different schedule from the other depending on the input data flow graph and constraints on the scheduling.

5.1.4. The maximal-scheduling and nonoverlap-scheduling

The maximal-scheduling algorithm schedules, starting from the most urgent operation, as many operations as possible in each time step as long as the longest path delay in a time step does not exceed the maximum stage-time limit. As the result, in the forward maximal schedule, each node is assigned to the **earliest** time step it can be assigned to. In the backward schedule, each node is assigned to the **latest** time step it can be assigned to. The maximal-scheduling algorithm assumes that each operation is to be performed on a different operator so that a fixed latency of 1 is possible. An example of a forward maximal schedule of the data flow graph of Figure 5.1-1 is shown in Figure 5.1-2.

Since the maximal-scheduling (either forward or backward) algorithm schedules as many operations as possible in each time step within the maximum stage-time limit, it produces the shortest schedule in terms of the number of time steps (refer to the proof of Lemma 3.2.1 in Chapter 3). Thus, from a maximal schedule, we can compute the absolute upper bound on the performance of a pipeline implementation of a data flow graph. The absolute fastest speed of a pipeline implementation of the data flow graph of Figure 5.1-1, according to the maximal-schedule of Figure 5.1-2, is 10^7 initiations of computation tasks per second (1/100 (nsec.)).

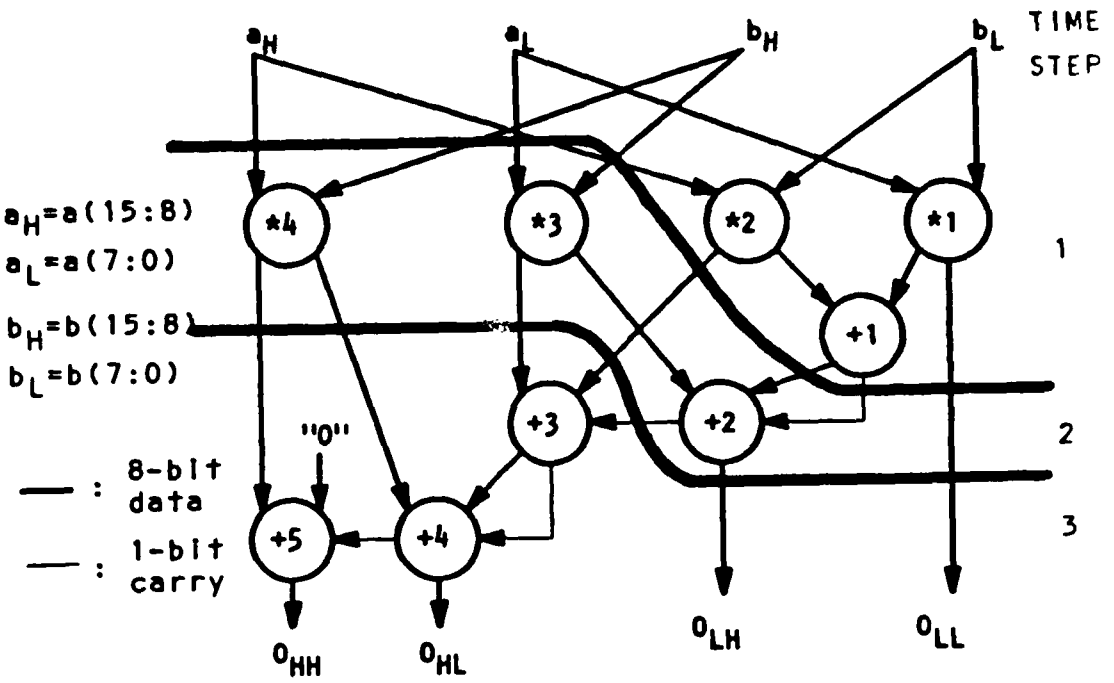
The maximal-schedule also gives the upper bound on the cost. If any other schedule results in a more expensive implementation than the maximal schedule, we would choose the maximal-schedule instead.



Note: 1. The operation-operator assignments and operator delays are the same as in Figure 5.1-1.
2. The maximum stage-time limit is 100 nsec.
3. Latch delays are assumed to be zero.

Figure 5.1-2: A forward maximal-schedule for the data flow graph of Figure 5.1-1.

The nonoverlap-scheduling algorithm schedules, also starting from the most urgent operation, as many operations as possible as long as the longest execution time of a time step does not exceed the maximum stage time limit and the resource requirement of each time step does not exceed the total available set of modules. Note that resources required can be shared across time steps, since time steps are never overlapped. An example of a backward nonoverlap-schedule of the data flow graph of Figure 5.1-1 is shown in Figure 5.1-3.



Note: 1. The operation-operator assignments and operator delays are the same as in Figure 5.1-1.
2. Two multipliers and three adders are used.
3. The maximum stage-time limit is 150 nsec.
4. Latch delays are assumed to be zero.

Figure 5.1-3: A nonoverlap schedule for the data flow graph of Figure 5.1-1.

According to the nonoverlap schedule of Figure 5.1-3, the speed of a non-pipelined implementation with two multipliers and three adders is 2,222,222 initiations per second (1/450 (nsec.)).

If a pipeline schedule at the same cost results in a slower speed than the nonoverlap-schedule, then we must discard the pipeline schedule.

5.2. The Feasible-scheduling Algorithm

The feasible-scheduling algorithm schedules a data flow graph with a fixed clock cycle (or the maximum allowed stage time), a fixed latency, and a fixed set of modules. This is the main algorithm for pipeline scheduling.

5.2.1. Algorithm outline

The outline of the feasible-scheduling algorithm (for either forward or backward scheduling) is shown below. We assume that there are either no mutually exclusive operations or at least no conditional resource sharing. Scheduling with conditional resource sharing will be discussed later.

Algorithm Feasible-Schedule

1. Compute urgencies of the nodes (either forward or backward) in the data flow graph;

2. Sort the nodes in non-decreasing order of urgency;

3. timestep := 1;

4. FOR the next most urgent node in the sorted list DO

   IF (adding the node to the current time step
   does not violate the maximum stage-time limit)
   AND
   (the node can be assigned to some module
   without any resource conflicts)

   THEN
   add the node to the current time step;
   delete the node from the sorted list;

5. IF NOT (the last node) THEN go to 4.

6. timestep := timestep + 1;

7. IF NOT (empty sorted list) go to 4.

Run-time analysis: Let m be the number of nodes in a data flow graph. The computation of urgencies of nodes require a breadth-first graph traversal, which takes $O(m^2)$ time steps. Sorting the nodes by urgency measure takes $O(m^2 log m)$, using the merge-sort. The actual scheduling loop (Step 4) checks at most n nodes at each iteration. This loop iterates at most S times where S is the number of time steps in the schedule, which is always less than n. Therefore, the run-time of this algorithm is determined by the merge-sort and is $O(m^2 log m)$.

We will illustrate this scheduling procedure through an example. At the same time, we will also show how urgency measures are used in scheduling together with design constraints.

Suppose that we have chosen two multipliers and three adders for the scheduling of the data flow graph of Figure 5.1-1. According to Theorem 4.3.5, the minimum possible fixed-latency is $\max\{\lceil\frac{4}{2}\rceil, \lceil\frac{5}{3}\rceil\} = 2$. We want to schedule the data flow graph with the minimum possible fixed-latency, 2, and a clock cycle of 150 nsec. We assume that the latch and multiplexer delays are zero. Figure 5.2-1 shows (a) a forward schedule and (b) a backward schedule. The procedure of the forward scheduling is as follows:

1. The nodes are sorted in non-increasing order of urgencies,
   (*1 *2 *3 +1 *4 +2 +3 +4 +5).

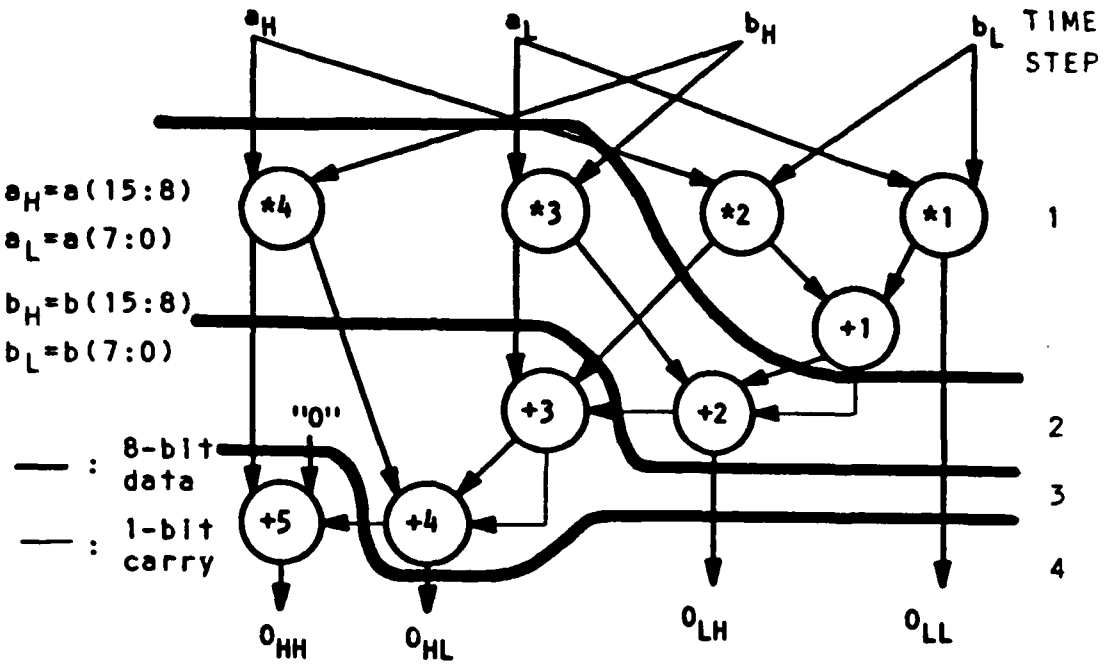
2. Get time step 1 from (*1 *2 *3 +1 *4 +2 +3 +4 +5).

   a. *1 and *2 are assigned to time step 1. *3 cannot be assigned to time step 1 since there are only two multipliers.

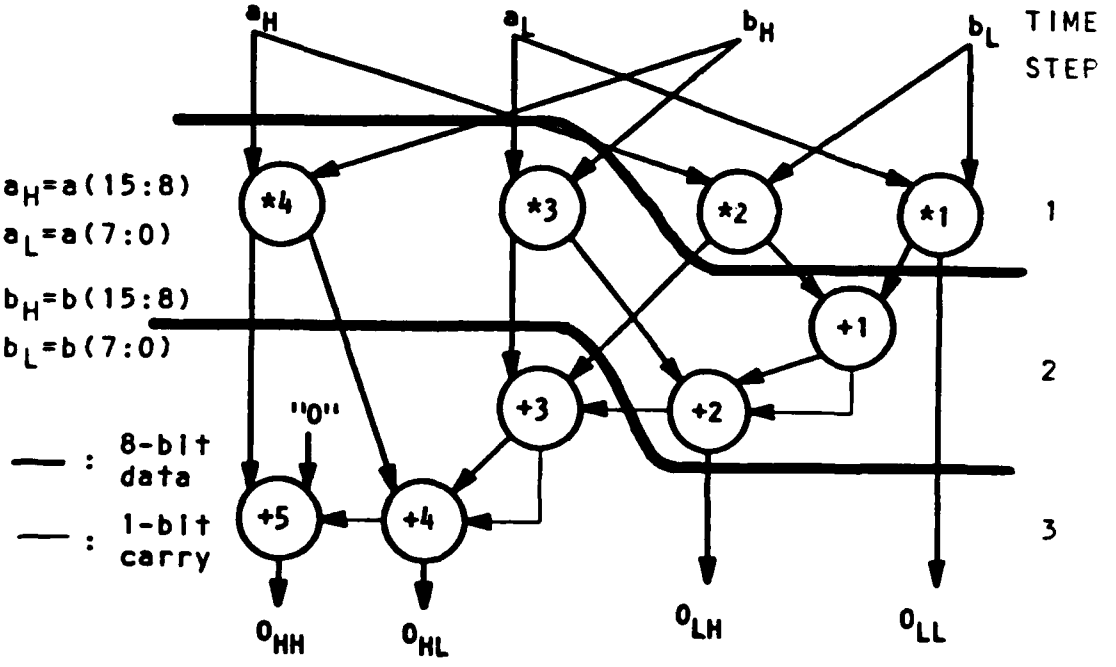
   b. +1 is added to time step 1. No more nodes can be added to time step 1 due to either the stage-time limit 150 nsec. or the number of available operators.

3. Get time step 2 from (*3 *4 +2 +3 +4 +5).

   a. *3 and *4 are assigned to time step 2. +2 is also assigned.



(a) A forward feasible-schedule



(b) A backward feasible-schedule

Figure 5.2-1: Examples of feasible-scheduling.

b. +3 cannot be assigned to time step 2 due to the stage time limit, i.e., the path delay through *3, +2, and +3 is 200 nsec. +4 and +5 cannot be assigned for the same reason.

4. Get time step 3 from (+3 +4 +5).

   a. +3 and +4 are assigned to time step 3.

   b. +5 cannot be added due to a resource conflict. Since the fixed latency is 2, time steps 1 and 3 are overlapped and the total number of adders needed in time steps 1 and 3 is already 3 for +1, +3 and +4 (refer to Theorem 4.3.3).

5. +5 is put in time step 4.

As shown in Figure 5.2-1, forward and backward schedules even with exactly the same design constraints result in different schedules. In general, no polynomial-time algorithms can guarantee optimality of the schedule. For this reason, it is desired that, for any polynomial-time scheduling algorithm, we use both approaches and choose the better schedule.

5.2.2. Resource allocation table

As we have seen in the previous scheduling example, resource conflict checking must be done for both a time step and between overlapping time steps. For scheduling of large data flow graphs, we need an efficient mechanism for resource conflict checking. The table shown in Figure 5.2-2, which can be used for both cases of resource conflict checking, is called a resource allocation table or simply an **allocation table**.

The number of rows in an allocation table corresponds to the number of operators. For the allocation table of Figure 5.2-2, there are one type-1 module and two type-2 modules, etc. The number of columns corresponds to the number of sets of overlapping time steps. As we have seen by Theorem 4.3.3 and Corollary 4.3.4, for a schedule with a fixed latency of l, there are at

\ time / steps	$\{F_{i+kl}\}$					
resource \	i=1	i=2	i=3	...	i=l	
module 1	OP1	OP5		...		
	OP2	OP4		...		
module 2		OP6		...		
				...		
.	.	.	.		.	.
.	.	.	.		.	.
				...		
module m	OP3			...		

Figure 5.2-2: The resource allocation table.

maximum l such overlapping sets of time steps, i.e., for a schedule of n time steps, F_1 through F_n, the sets of overlapping time steps for a fixed latency of l are $\mathcal{F}_i$ through $\mathcal{F}_l$ where

$$\mathcal{F}_i = \{F_{i+kl}\},\ 0 \leq k \leq \lfloor \frac{n-1}{l} \rfloor$$

Each cell in an allocation table represents an instance of usage of the corresponding type of module. Thus, in each cell, either only one unconditional operation or a set of mutually-exclusive operations can be assigned. Note that when there are multiple modules of the same type, the number of filled-in cells of a type of module represents only how many of them are needed during one clock cycle, and there is no need to freeze individual module assignment of the operations yet.

Figure 5.2-3 shows the state changes of the allocation table during the

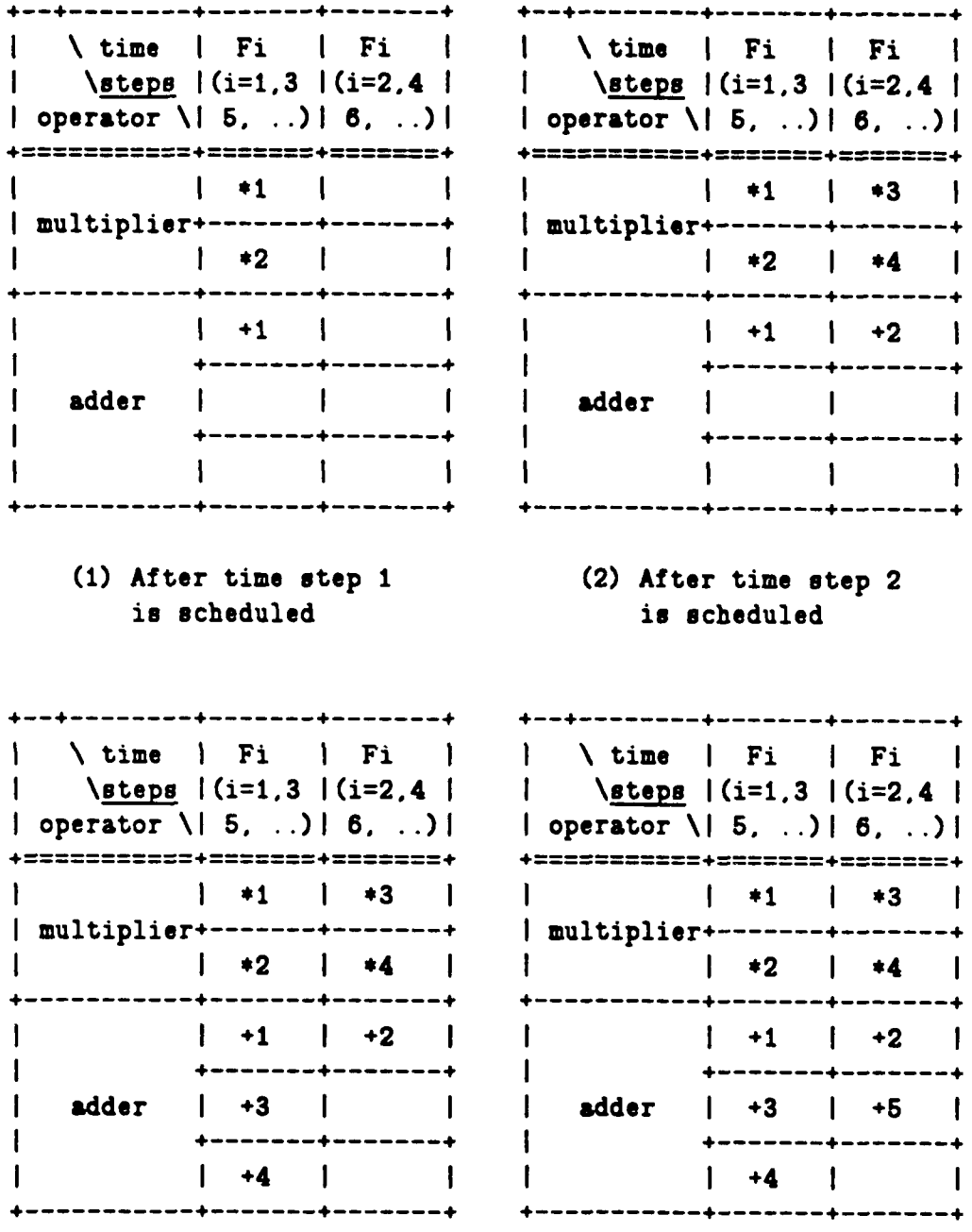


Figure 5.2-3: An example usage of the allocation table during the forward scheduling shown in Figure 5.2-1-(a).

forward scheduling shown in Figure 5.2-1-(a). An example detection of possible resource conflicts between overlapping time steps can be found in the allocation table state (3) in Figure 5.2-3. After +4 is added to time step 3 or F3, the first column of the allocation table is full. Thus, +5 cannot be added to F3 even though adding it to F3 does not violate the 150 nsec. stage-time limit.

Using this allocation table, we can rewrite Step 4 of the algorithm, Feasible-Schedule, as follows:

4. FOR the next most urgent node in the sorted list DO

IF (adding the node to current time step does
not violate the maximum stage-time limit)
AND
(there is a cell in column[timestep *mod* latency]
and row[needed module] in the allocation table
which is *either* empty *or* filled with nodes all
mutually exclusive with the node to be added)

THEN
add the node to current time step;
delete the node from the sorted list;

5.2.3. Resource sharing between mutually exclusive operations

In this section, we discuss conditional resource sharing between mutually exclusive operations during the feasible-scheduling. We focus on two aspects of conditional resource sharing:

- forcing conditional resource sharing due to limitations on the number of available modules and the fixed latency, and
- determination of when to force conditional resource sharing.

We discuss these aspects through an example of a forward feasible scheduling of the data flow graph of Figure 4.4-1, as shown in Figure 5.2-5.

Suppose that all the additions are to be performed by adders with delay time 100 nsec. and subtractions by subtractors with delay time 100 nsec. We want a schedule with a fixed-latency of 3 and a clock cycle of 100 nsec. (i.e., at maximum, we can initiate a new computation task every 300 nsec.). According to Lemma 4.5.3, the maximum possible numbers of additions and subtractions actually performed during an execution of a computation task are 6 and 5, respectively. Accordingly, by Theorem 4.3.5, we choose $\lceil \frac{6}{3} \rceil = 2$ adders and $\lceil \frac{5}{3} \rceil = 2$ subtractors.

As shown in the allocation table of Figure 5.2-4, we can use the adders and subtractors only six times each, although we have seven subtraction nodes and eight addition nodes in the data flow graph of Figure 5.2-5. Therefore, at some point, we have to *force conditional resource sharing* in order to satisfy the resource limit with the chosen latency.

Now we illustrate this through a forward feasible-scheduling of the data flow graph of Figure 5.2-5, with the constraints mentioned above. The forward urgency measures of the operations is shown below.

Node	+1	-3	+2	-1	-2	+3	+6	+4
Forward Urgency	500	400	300	300	300	300	300	200

Node	-4	+5	-5	-6	+7	-7	+8
Forward Urgency	200	200	200	200	100	100	100

Figure 5.2-4 shows the allocation table for this scheduling. It shows the status of the resource allocation in the middle of scheduling time step 2.

\ time / \ steps / operator	Fi (i=1,4,7, ..)	Fj (j=2,5,8, ..)	Fk (k=3,6,9, ..)
subtractor	-1	(-3,-2)	
adder	+1 +2	+3	

Figure 5.2-4: An allocation table with two adders and two subtractors, and a fixed latency of 2.

First, +1, +2, and -1 are assigned to time step 1 as shown in Figure 5.2-5. The remaining nodes in the order of forward urgency are (-3 -2 +3 +6 +4 -4 +5 -5 -6 +7 -7 +8). No more nodes can be added to time step 1 due to the stage-time limit of 100 nsec.

Next, -3 and -2 are assigned to time step 2. They are mutually exclusive to each other, thus are put in the same cell. Note that there are only four subtraction nodes left to be scheduled and there are four empty cells of subtractors in the allocation table. Therefore, all the subtraction nodes can be scheduled without any resource conflicts.

The remaining nodes are now (+3 +6 +4 -4 +5 -5 -6 +7 -7 +8). The next most urgent node is +3. However, if +3 alone is assigned to an empty cell of adders, there will be only three empty cells of adders left in the allocation table. The remaining addition nodes are (+6 +4 +5 +7 +8). Even if the mutually exclusive operations +5 and +6 are assigned to one cell, there must be at least 4 empty cells of adders. Therefore, assigning +3 alone in an empty cell of adders is not possible. The only feasible allocation is assigning the three

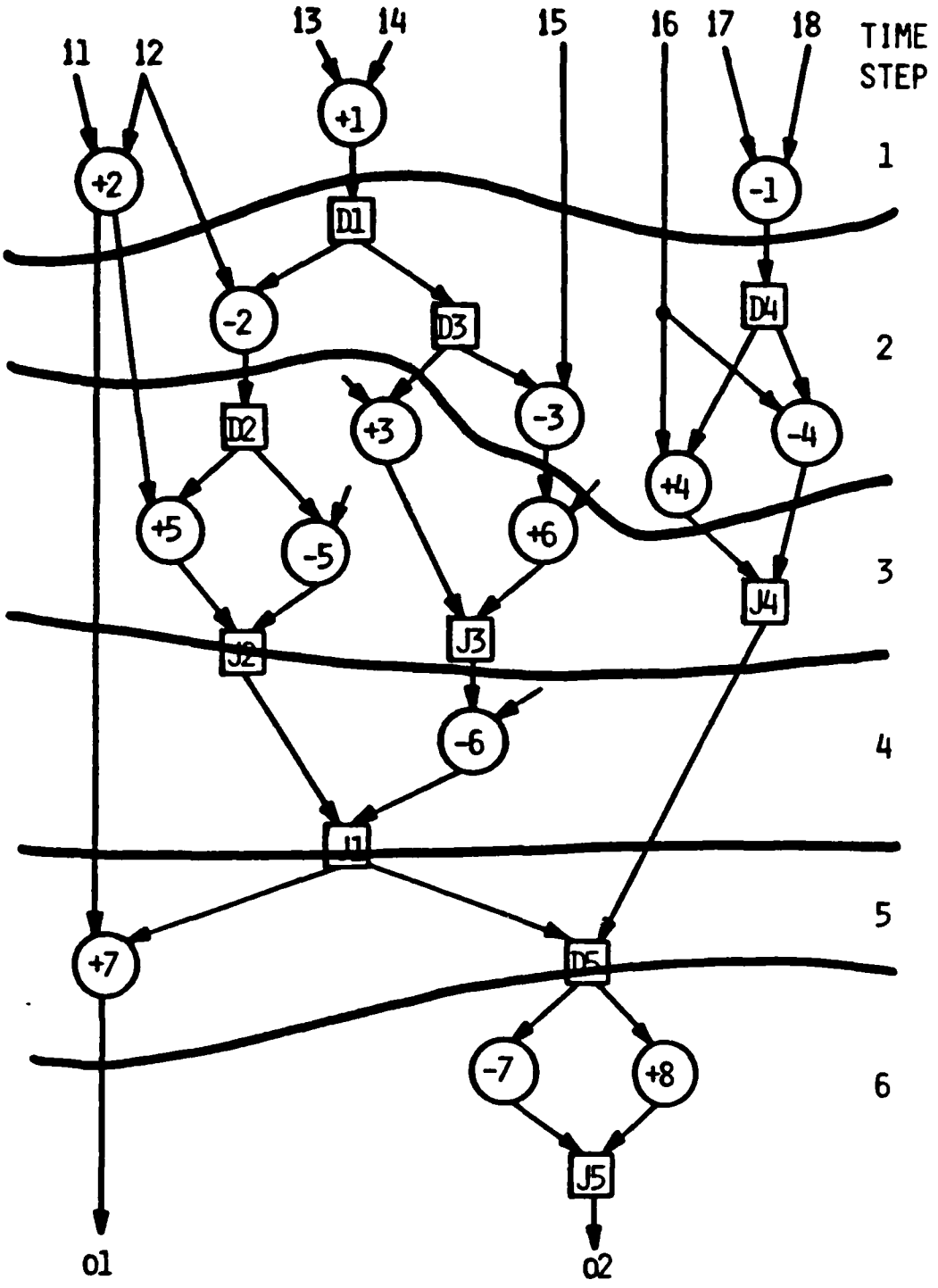


Figure 5.2-5: The data flow graph of Figure 4.4-1 scheduled.

mutually exclusive operations, +3, +6, and +5, all in one cell of adders. Therefore, we have to force conditional resource sharing for these three addition nodes. However, assigning +6 to time step 2 will make the stage time 200 nsec. (through -3 and +6). Therefore, +3, +6, and +5 must be assigned to some later time step all together.

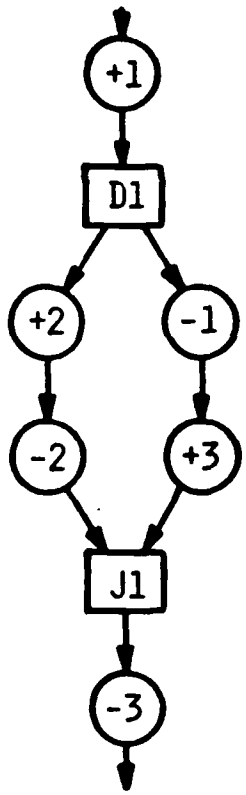
\ time / steps / operator	Fi (i=1,4,7, ..)	Fj (j=2,5,8, ..)	Fk (k=3,6,9, ..)
subtractor	-1	(-3,-2)	-5
	-6	-4	-7
adder	+1	+4	(+3,+6,+5)
	+2	+7	+8

Figure 5.2-6: The completed resource allocation.

Figures 5.2-5 and 5.2-6 show the completed schedule and resource allocation.

Forcing conditional resource sharing may cause a deadlock in the algorithm. As an example, we consider scheduling the data flow graph of Figure 5.2-7-(a).

Suppose that we have chosen one adder and one subtractor both of which have 100 nsec delay time. In this data flow graph, each conditional execution path has two subtraction nodes (either -2 and -3, or -1 and -3) and two addition nodes (either +1 and +2, or +1 and +3). Thus, according to Lemma 4.5.3, we want a schedule with latency 2. Figure 5.2-7-(b) shows an allocation table for this scheduling. Suppose that we want the stage time to be 100 nsec.



(a) A data flow graph.

\ time / operator	Fi (i=1,3 5, ..)	Fi (i=2,4 6, ..)
adder	+1	
subtractor		

(b) Resource allocation table for a forward feasible-scheduling with one adder and one subtractor with a fixed latency of 2.

Figure 5.2-7: An example of a deadlock due to forced conditional resource sharing.

As shown in the data flow graph and the allocation table, only +1 is assigned to time step 1 due to the stage-time limit. +2 and -1 are the next most urgent nodes. In the allocation table, there is only one empty cell for the mutually-exclusive addition nodes, +2 and +3. For the subtraction nodes -1, -2 and -3, there are two empty cells. Therefore, for both the addition and subtraction nodes, conditional resource sharing must be forced (between +2 and +3, and between -1 and -2). However, due to the stage-time limit, neither of the mutually-exclusive operation pairs can be added to time step 2. In order for -1 and -2 to be assigned to the same time step, +2 must be assigned to some earlier or the same time step, which is not possible due to either the resource

constraint or the stage-time limit. Also, for +2 and +3, -1 must be assigned to some earlier or the same time step. Therefore, there is a **deadlock**.

In order to resolve a deadlock, either more modules than the minimum possible number must be used, and/or the stage-time limit must be increased.

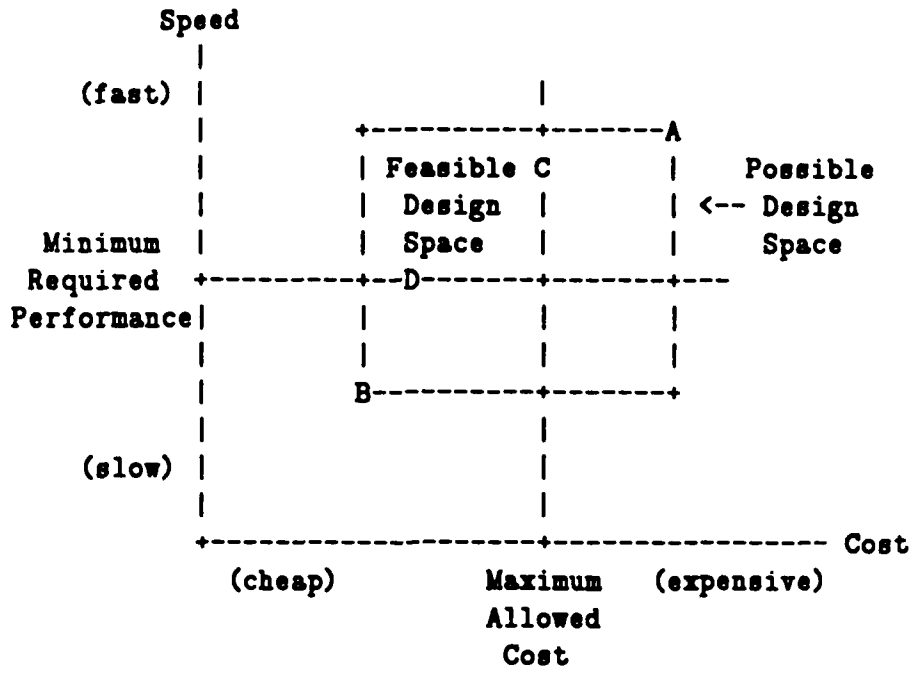
5.3. Scheduling with Speed and Cost Constraints

In this section, we show how the scheduling and resource allocation techniques we discussed can be used in synthesizing a near-optimal pipeline. We divide the design tasks into two cases depending on the design constraint:

- design the fastest pipeline within the cost constraint, or
- design the cheapest pipeline that satisfies the minimum required performance.

5.3.1. Design-space boundaries

Figure 5.3-1 shows the design space boundary for a pipeline design from a data flow graph. In the figure, the absolute fastest point, A, is determined by the maximal-scheduling. The absolute cheapest point, B, is determined by the nonoverlap scheduling with the absolute minimum set of modules (i.e. one for each type). If the total cost is limited, point C is the optimal design. If the minimum required performance is given, point D is the cheapest design. The feasible design space is enclosed in the rectangle of the points C and D. While the scheduling is repeated with different cost and performance constraints, we only need to search the feasible space inside the possible design space.



A: The absolute fastest design
B: The absolute cheapest design
C: The fastest design within cost constraint
D: The cheapest design satisfying minimum required performance

Figure 5.3-1: Design space boundaries.

5.3.2. Synthesis with a cost constraint

The feasible-scheduling algorithm we discussed uses a fixed set of modules for scheduling and resource allocation. Thus, if simply the total cost of a design is limited, we want to choose an optimal number of each type of required module within the cost constraint (we assume that the choice of which types of modules to use is fixed). However, the optimality of module selection can only be known after a scheduling and resource allocation. If an optimal design is desired, we need to compare the result of all possible sets of modules within the cost constraint. To avoid this time-consuming iteration, it is

desirable that the synthesis procedure analyze the results of scheduling and predict a better set of modules for the next scheduling.

In this section, we outline such a synthesis procedure. We first define some variables for the convenience of discussion.

N_i	The maximal number of type-i operations which must be performed during an execution of a computation task.
M_i	The number of type-i modules chosen to perform type-i operations..
C_i	The cost for a type-i module.
C_{total}	The maximum allowed cost for a design.
l	A chosen latency.

Procedure Cost-Constrained-Synthesis

1. Compute the minimum possible latency and the number of modules such that $\sum_i \lceil \frac{N_i}{l} \rceil \cdot C_i \leq C_{total}$
2. FOR every possible stage time,[21]
   do forward and backward feasible-scheduling;
   put the schedules in the solution set;
3. FOR every schedule in the solution set,
   A. compute the total cost of design by adding the latch and multiplexer cost to the operator cost;
   B. compute the performance of the design considering the resynchronization rate;

[21]For the computation of all possible stage times, see the run time analysis of the OPART algorithm in Section 3.2.3.

4. IF (there are any solutions within the cost constraint)
   THEN
      A. choose the fastest design[22] within the cost constraint;
      B. Check if there is any possibility of increasing the performance of the fastest design by adding more modules within the cost constraint.

         IF (yes)
            THEN add more module and go to 2;
            ELSE go to 5.

   ELSE
      {*If there is no solution at all, choose*}
      {*a cheaper set of modules and restart *}
      increment l and recompute the minimum necessary number of modules;
      go to 2;

5. Report the fastest design within the cost constraint as the solution;

   Choose the cheapest design which is faster than the solution, and report it as an alternative design;

In Step 4-B, we need to analyze the schedule first and determine whether there is a chance of improving the schedule by increasing certain modules. In order to do this, we rewrite the Step 4 of the algorithm Feasible-Schedule presented in Section 5.2.1 as follows:

[22] A design includes a schedule with resource allocation, latency, clock cycle, stage latch locations, and total cost and performance estimation.

4. FOR the next most urgent node in the sorted list DO

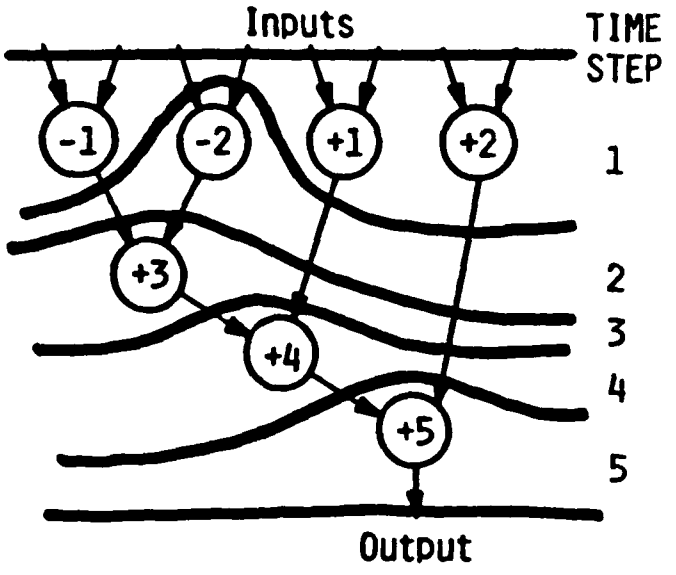
```
IF    (adding the node to current time step does
          not violate the maximum stage-time limit)
                    AND
      (there is a cell in column[timestep mod latency]
       and row[needed module] in the allocation table
       which is either empty or filled with nodes all
       mutually exclusive with the node to be added)
  THEN
      add the node to current time step;
      delete the node from the sorted list;
  ELSE IF (adding the node to current time step does
          not violate the maximum stage-time limit)
      THEN
         increment no_module count;
```

The counter **no_module** indicates how many times the operations assigned to a certain type of module were delayed to some later time steps only due to the resource limit. Therefore, if we increase the number of modules with positive no_module counts, we might be able to shorten certain schedules. If there is more than one type of module with a positive no_module count, we use the following tie-breaking rules:

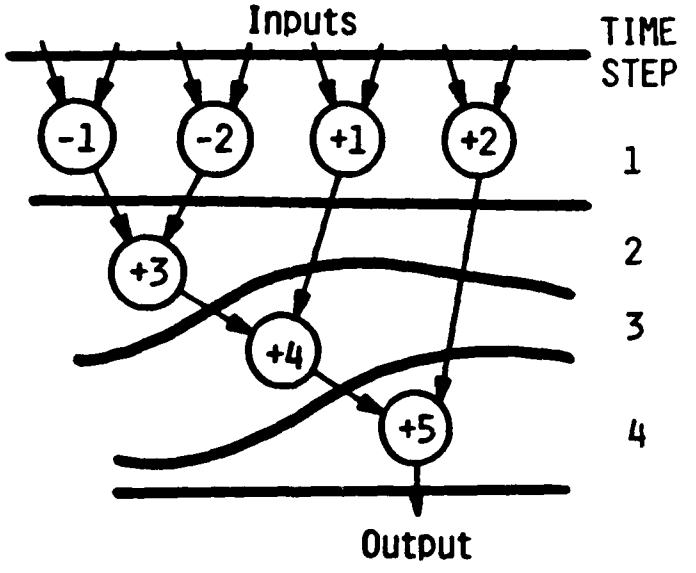
1. choose the largest no_module count, then
2. the cheapest module, and then
3. the module required by the most urgent operation.

This is illustrated in Figure 5.3-2.

During the scheduling of Figure 5.3-2-(a), only the subtraction node -2 is delayed since there is only one subtractor. The resulting schedule has five time steps. In (b), two subtractors are used and the resulting schedule has four time steps. According to Corollary 4.3.2, for a sequence of computation tasks with any resynchronization overhead, schedule (b) is always faster than schedule (a).



(a) Forward feasible schedule with one subtractor and three adders (latency = 2).



(a) Forward feasible schedule with two subtractors and three adders (latency = 2).

Note: Adders and subtractors have the same delay time, and the stage-time limit is set to one operator delay.

Figure 5.3-2: Performance improvement by adding more modules.

5.3.3. Synthesis with performance constraints

If the minimum required performance is given as a design constraint, we would like to synthesize the cheapest pipeline that meets the performance constraint. We start scheduling with the maximum possible latency, which requires the minimum set of modules. If no feasible solution is found, the latency is decremented. Until a solution is found, the number of modules is minimum for each chosen latency. Once a solution is found with the minimum number of modules for a latency, the latency is incremented and the procedure Cost-Constrained-Synthesis is called once to search for a cheaper solution. This is necessary since the solution found as such uses the minimum number of modules for the latency used, say l, although there may be a cheaper solution with latency $l+1$ but not with the minimum number of modules.

The outline of the **Performance-Constrained Synthesis** procedure is shown below. The definitions of the variables are the same as in the Cost-Constrained-Synthesis procedure.

Procedure Performance-constrained-Synthesis

1. Compute the maximum potentially possible latency such that $\frac{1}{l \cdot (\textit{minimum possible stage time})} \leq (\textit{desired average initiation rate})$.

2. Compute the minimal set of modules for latency l.

3. Choose all the feasible stage times such that $\frac{1}{l \cdot (\textit{stage time})} \leq (\textit{desired average initiation rate})$.

4. FOR every feasible stage time,
   do forward and backward feasible-scheduling;
   put the schedules in the current solution buffer;

5. FOR every schedule in the current solution buffer,
   A. compute the total cost of design by adding the latch and multiplexer cost to the operator cost;
   B. compute the performance of the design considering the resynchronization rate;

6. IF (there are any new solutions satisfying the minimum required performance)
   THEN
      A. append the current solution buffer to the solution set;
      B. increment the latency and go to 2;
         {*A solution found. Try cheaper designs*}

   ELSE IF (there is no feasible solution yet)
      decrement the latency and go to 2;
   ELSE {*a solution was found before but not with current latency*}
      go to 7;

7. Choose the cheapest design satisfying the performance constraint as the candidate for the solution;

8. Call Cost-constrained-Synthesis with the cost for the solution candidate (looking for a cheaper design) and append the result to the solution set;

9. Report the cheapest design satisfying the performance constraint as the solution;

5.4. An Exhaustive Algorithm for Pipeline Synthesis

In this section, we examine the possibility of using an exhaustive algorithm for optimal scheduling and resource allocation of a data flow graph to be pipelined, while satisfying cost and performance constraints.

For a data flow graph to be scheduled for pipelining, if the maximum stage-time limit is given, the shortest possible schedule with the smallest number of times steps is determined by the maximal schedule (either forward or backward), as discussed in Section 5.1.4. If the the difference in the number

of time steps between the maximal schedule and the current best feasible schedule is large, there is a possibility of existence of a shorter feasible schedule. (However, note that the maximal-scheduling algorithm assumes infinite resources. There may not be any better feasible schedule.) We present an exhaustive algorithm to find such a shorter schedule.

The inputs to the exhaustive algorithm are

- the maximum stage time limit,
- the latency,
- the number of available modules, and
- the current best feasible schedule.

The exhaustive scheduling algorithm takes the best feasible schedule produced by the feasible-scheduling algorithm as a good lower bound on performance to prune the search space. The outputs are the shortest feasible schedule with resource allocation, the total cost including the stage-latch cost, and performance estimation.

We first outline the algorithm, and analyze each step. The outline of the exhaustive scheduling and resource allocation algorithm is as follows:

1. Do forward and backward maximal-scheduling, and determine the minimum possible number of time steps, S_{min}.
2. Set the desired number of time steps, S_d, less than that of the current best feasible schedule by one.
3. Compute the earliest and latest possible time steps for every node in the data flow graph, according to the maximal schedules and S_d.
4. Enumerate all possible time-step assignments for the nodes (schedules) within the earliest and the latest time-step ranges of the nodes. For each configuration, check if the stage-time is within the limit and the resource allocation is feasible.

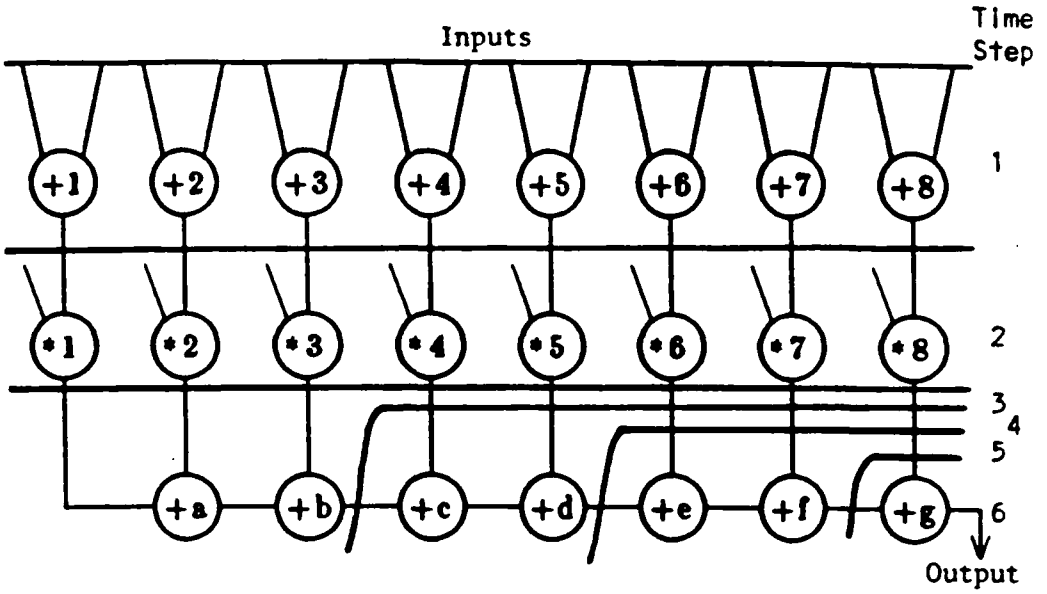
5. If any solution is found, then report it. If still $S_d > S_{min}$, then decrement S_d and go to Step 3.

6. If either no solution is found or $S_d = S_{min}$, STOP.

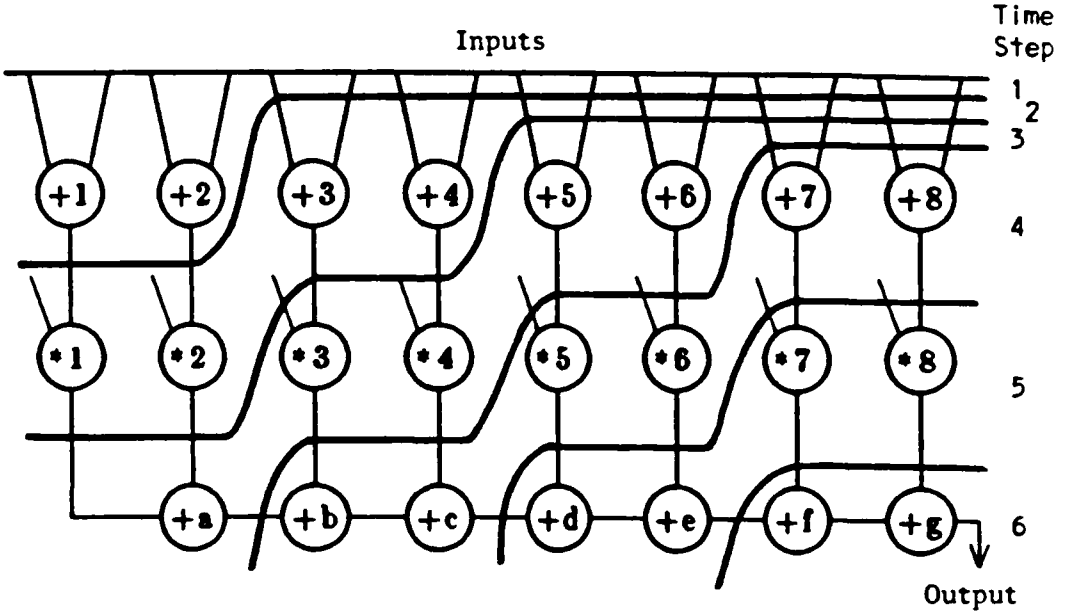
First of all, we show how the earliest and the latest possible time steps for each node are determined by the maximal schedules and the desired number of times steps in the schedule. As we discussed in Section 5.1.4, the time step of each node in the forward maximal schedule is the earliest time step the node can be scheduled, and the time step of each node in the backward schedule is the latest time step the node can be scheduled. &&& Therefore, for a desired number of time steps, S_d, the possible range of time steps for node i, R(i), with the earliest possible time step, E(i), and the latest possible time step, L(i), is $[E(i) \sim L(i)+(S_d-S_{min})]$, inclusively. For example, in Figure 5.4-3, since the S_d is the same as the S_{min}, which is 6, the time step range of node i is $[E(i) \sim L(i)]$. If S_d is 7, then the range of every node will be augmented by 1, e.g., the range for the +1 node will be $[1 \sim 2]$ and for the +3 node will be $[1 \sim 3]$.

Now we illustrate the algorithm with a 16-point digital FIR filter example. In this example, we assume that multiplications (*) are performed by multipliers with delay time 80 nsec., and additions (+) are performed by adders with delay time 40 nsec. The latch delays are 20 nsec. The maximum stage time limit is set to 100 nsec. Figure 5.4-1 shows the maximal schedules of the data flow graph, and Figure 5.4-2 shows the backward feasible schedule.

As shown in Figure 5.4-2, the backward feasible schedule has one more time step than the maximal schedules, i.e., 7 time steps. According to Corollary 4.3.2, the average initiation interval for this feasible schedule is $(1+2\rho)\cdot 300$ nsec., where ρ is the resynchronization rate. If we can reduce the number of time steps by one, the average initiation interval becomes $(1+\rho)\cdot 300$



(a) The forward maximal schedule.



(b) The backward maximal schedule.

Figure 5.4-1: The maximal schedules for a 16-point FIR digital filter with stage time limit 100 nsec.

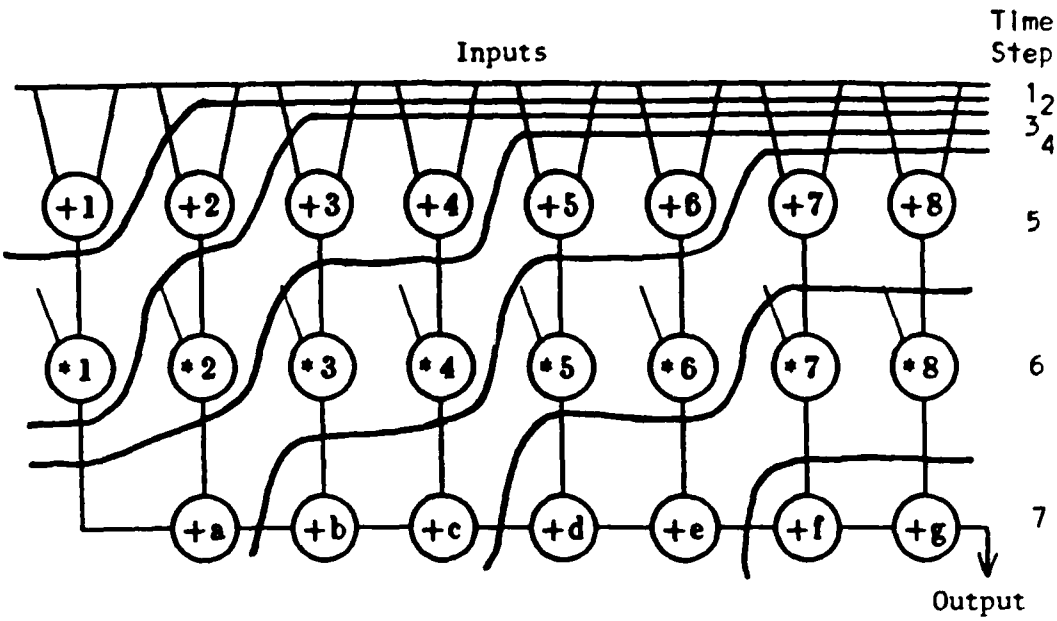


Figure 5.4-2: A backward feasible schedule for the digital filter with 3 multipliers and 5 adders, latency 3, and stage-time limit 100 nsec.

nsec., which is shorter than the design with 7 time steps, if there is any resynchronization overhead.

In Figure 5.4-3, the possible time ranges for each node according to the maximal schedules are shown. Since the desired number of time steps for a better feasible schedule is the same as that of a maximal schedule, 6, the time range for a node is from the earliest to the latest in the maximal schedules.

Using the information in Figure 5.4-3, the exhaustive algorithm enumerates all possible schedule vectors:

N(i)	E(i)	L(i)	R(i)
+1	1	1	1
+2	1	1	1
+3	1	2	1 - 2
+4	1	2	1 - 2
+5	1	3	1 - 3
+6	1	3	1 - 3
+7	1	4	1 - 4
+8	1	4	1 - 4
*1	2	2	2
*2	2	2	2
*3	2	3	2 - 3
*4	2	3	2 - 3
*5	2	4	2 - 4
*6	2	4	2 - 4
*7	2	5	2 - 5
*8	2	5	2 - 5
+a	3	3	3
+b	3	4	3 - 4
+c	4	4	4
+d	4	5	4 - 5
+e	5	5	5
+f	5	6	5 - 6
+g	6	6	6

* The desired number of time steps is 6.

N(i): a node.

E(i): the earliest possible time step for N(i), derived from the forward maximal schedule of Fig. 5.4-1-(a).

L(i): the latest possible time step for N(i), derived from the backward maximal schedule of Fig. 5.4-1-(b).

R(i): the possible time step range for N(i) for feasible schedules with the same number of time steps as the maximal schedule.

Figure 5.4-3: Time step ranges of the nodes of the FIR filter example.

(1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 4 4 5 5 6)
(1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 4 4 5 6 6)
(1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 4 5 5 5 6)

.
.
.

(1 1 2 2 3 3 4 4 2 2 3 3 4 4 5 5 3 4 4 5 5 5 6)
(1 1 2 2 3 3 4 4 2 2 3 3 4 4 5 5 3 4 4 5 5 6 6)

where the n-th element in every vector represents the time step assignment of

the n-th node. The schedule vector can be considered to be a cascade of modulo counters, where the i-th stage is a modulo-|R(i)| counter, where $|R_i|$ is the length of the time-step range of node i. For each schedule vector configuration, we must check the data precedence between nodes, the stage-time limit and the feasibility of resource allocation.

Feasibility Checking

Data precedence checking can be performed by comparing the time steps of every node with its children's, which requires at most n comparisons for each node. Thus, for each configuration, $O(n^2)$ comparisons are needed. A node must be assigned to an earlier than or at least equal time step to any of its child nodes.

The stage time limit must also be checked. Let m(i) be the delay time of the module that node i is assigned to. The longest path of a time step can be computed as follows:

1. For every node, i, set weight(i) := m(i);

2. FOR every node, i,
   (in non-decreasing order of forward urgency) DO

   FOR every child, j, of i **in the same time step** DO
   IF weight(i) + m(j) > weight(j)
   THEN weight(j) := weight(i) + m(j);

3. Find the maximum weight(i) and add the latch delay to it;

No child node can be more urgent than its parents, and thus, no child node is weighted before all its parent nodes are weighted. Therefore, the weighting process proceeds from the input side of the data flow graph to the output side. This also takes $O(n^2)$ time steps since there are at most n children for each node.

The feasibility of the resource allocation can be checked by filling in the allocation table we discussed in the previous chapter, which also takes $O(n^2)$ time steps.

Skipping Schedule Vectors

We fill in the resource allocation table from the left-most node to the right as the nodes appear in the schedule vector. Suppose that the i-th node cannot be scheduled in the time step as specified by the configuration vector due to the resource limit. Then, as long as the elements on the left side of the i-th element in the configuration vector, the first through (i-1)-th, are not changed, the i-th node cannot be assigned. Therefore, we do not need to check any other configuration with the same values of the elements from the first through (i-1)-th. We can reset the (i+1)-th through the last elements to the earliest time steps and increment the configuration vector from the i-th element.

A solution is found when the schedule vector is

(1 1 1 2 2 2 3 3 2 2 3 3 4 4 4 5 3 4 4 5 5 6 6)

and is shown in Figure 5.4-4.

We analyze the run-time complexity of this algorithm based on an implementation in Franz LISP. Let n be the number of nodes in a data flow graph. The longest range of the time steps of a node is O(n), since there can be at most O(n) number of time steps in a schedule. Therefore, the schedule vector is incremented at most $O(n^n)$ times. For each schedule vector, the feasibility of the stage time and the resource allocation is checked in $O(n^2)$ time steps as we have discussed above. Therefore, the run-time complexity of this algorithm is $O(n^{n+2})$.

However, in practice, the range of possible time steps for a node is much less than n, as shown in Figure 5.4-3. Also, as we discussed just above, not all the configurations need to be checked. Also, a solution can possibly be found

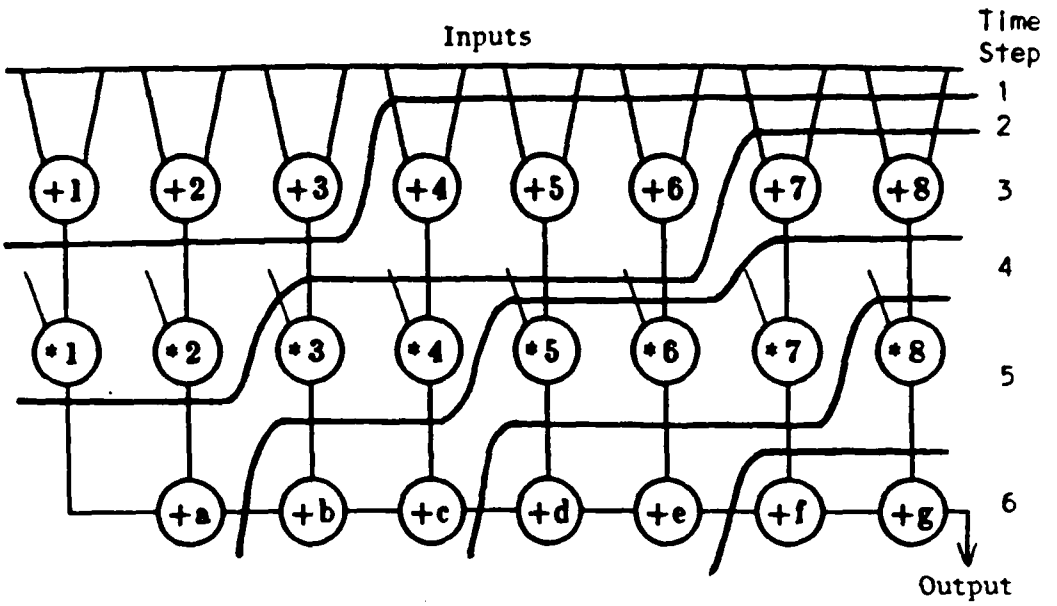


Figure 5.4-4: An optimal feasible schedule for the digital filter.

at an earlier schedule vector. For the example FIR filter shown above, the actual run-time of the LISP procedure on the VAX/750 is less than a CPU hour for most cases of module sets and stage time limits.

5.5. A Synthesis Example

The theory and algorithms for pipeline synthesis we discussed in Chapter 4 and in this chapter are implemented in Franz LISP, and integrated in the main synthesis procedure, **Sehwa**. We demonstrate this program with an example. We use the data flow graph of Figure 5.2-5 as an example task to be pipelined, which we have seen in Section 5.2.3. A brief description of the inputs and design goals is followed by the output listing of the program.

As shown in the output listing, the data flow graph is input in a LISP-list format, consisting of a list of node descriptions and a list of edge descriptions. A node description is a list consisting of three elements, a *name*, *function*, and

the *bitwidth* of the operation. An edge description is also a list and consists of five elements, a *name*, a *source node*, a *sink node*, *bitwidth*, and the *name of the value* it is carrying.

Module selection for each type of function is done manually. The module description format is shown in the output listing. After the module selection is done and the latch timing information is given, the program outputs design boundaries, which we discussed in Section 5.3.1.

First, the cost-constrained-synthesis mode is selected and the maximum design cost is set to 8 unit costs. The expected resynchronization rate is set to 15%. The main procedure called the procedure, Cost-Constrained-Synthesis. The best solution found by the procedure, Cost-Constrained-Synthesis, has an effective initiation interval of 414.1 nsec. at the cost of 7.2 unit costs.

Next, we decided to make the effective initiation interval shorter than 400 nsec. Also, we increased the expected resynchronization rate to 20%. The solution found by the procedure, Performance-Constrained-Synthesis, has an effective initiation interval of 336.1 nbsec. at the cost of 9.2 unit costs.

On exit, the main procedure called the maximum-scheduling procedure with the same stage time (clock cycle) limit as the current solution, and compared the performance of the maximal schedule with that of the current solution. The main procedure decided that the performance difference is big and there is a chance of the existence of a better feasible schedule. The exhausitive-scheduling procedure is called with the same number of modules and the stage time limit with the current solution, although no better feasible schedule was found.

Output Listing

Note: In the schedule, conditional resource sharing (refer to Definition 4.4.1 in Section 4.4.2) is indicated by a pair of triangle brackets, e.g., <2(sub3) 1(sub2)>.

```
[1] lisp Sehwa.l
Franz Lisp, Opus 38.79
[load Sehwa.l]
-> (Sehwa)

   WELCOME TO USC-SEHWA PIPELINE SYNTHESIS PACKAGE!
        *** USC Design Automation Group ***
                  Nohbyung  Park

; ************************************************** ;
;  Input Processing and Data Structure Initialization ;
; ************************************************** ;
Input Data-Flow Graph (File Name)? dfg.c5

*** Input Data Flow Graph ***

Node List: <index (name function bitwidth)>
(  0  (sub1 sub 16)    1  (sub2 sub 16)    2  (sub3 sub 16)
   3  (sub4 sub 16)    4  (sub5 sub 16)    5  (sub6 sub 16)
   6  (sub7 sub 16)    7  (add1 add 16)    8  (add2 add 16)
   9  (add3 add 16)   10  (add4 add 16)   11  (add5 add 16)
  12  (add6 add 16)   13  (add7 add 16)   14  (add8 add 16)
  15  (D1 dist)       16  (D2 dist)       17  (D3 dist)
  18  (D4 dist)       19  (D5 dist)       20  (J1 join)
  21  (J2 join)       22  (J3 join)       23  (J4 join)
  24  (J5 join))

Edge List: <index (source destination bitwidth value)>
(  0  (i1 root add2 16 i1)          1  (i21 root add2 16 i2)
   2  (i22 root sub2 16 i2)         3  (i3 root add1 16 i3)
   4  (i4 root add1 16 i4)          5  (i5 root sub3 16 i5)
   6  (i61 root sub4 16 i6)         7  (i62 root add4 16 i6)
   8  (i7 root sub1 16 i7)          9  (i8 root sub1 16 i8)
  10  (se1 sub1 D4 16 se1)         11  (se2 sub2 D2 16 se2)
  12  (se3 sub3 add6 16 se3)       13  (se4 sub4 J4 16 se4)
```

```
 14  (se5 sub5 J2 16 se5)          15  (se6 sub6 J1 16 se6)
 16  (se7 sub7 J5 16 se7)          17  (ae1 add1 D1 16 ae1)
 18  (ae21 add2 add7 16 ae2)       19  (ae22 add2 add5 16 ae2)
 20  (ae3 add3 J3 16 ae3)          21  (ae4 add4 J4 16 ae4)
 22  (ae5 add5 J2 16 ae5)          23  (ae6 add6 J3 16 ae6)
 24  (ae7 add7 outport 16 ae7)     25  (ae8 add8 J5 16 ae8)
 26  (de11 D1 sub2 16 de1)         27  (de12 D1 D3 16 de1)
 28  (de21 D2 add5 16 de2)         29  (de22 D2 sub5 16 de2)
 30  (de31 D3 add3 16 de3)         31  (de32 D3 sub3 16 de3)
 32  (de41 D4 add4 16 de4)         33  (de42 D4 sub4 16 de4)
 34  (de51 D5 sub7 16 de5)         35  (de52 D5 add8 16 de5)
 36  (je11 J1 add7 16 je1)         37  (je12 J1 D5 16 je1)
 38  (je2 J2 J1 16 je2)            39  (je3 J3 sub6 16 je3)
 40  (je4 J4 D5 16 je4)            41  (je5 J5 outport 16 output))

 ****************************************
 *  WELCOME TO MODULE-SELECTION PHASE   *
 ****************************************
Do you want instructions (y/n)? y

          <INSTRUCTION FOR MODULE SELECTION>

This is an interactive and iterative module-selection routine.
Type-in a  module-description for each function one by one.

MODULE DESCRIPTION FORMAT:
<module>::=(<name><operation><bitwidth><cost><delay_time>)

Each module_name must be unique. The bitwidth and module delay
(nsec) are positive integers and the cost is positive real.

Ex1:   csadder8 add 8 0.5 30   -- An 8-bit carry-save adder
Ex2:   TI74284 mul 4 2.0 60    -- TI 4-bit binary multiplier

Type carriage-return (CR) to continue ...

    *** Function List ***

 (function  (node_indices))
   (sub (6 5 4 3 2 1 0))
   (add (7 8 9 10 11 12 13 14))
```

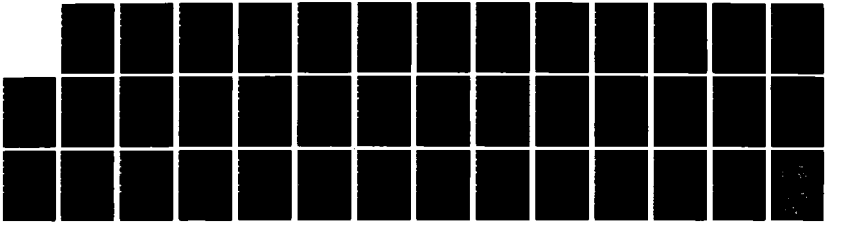
```
> Function:  sub,  Total number of nodes: 7
  Max. number of possible evaluations: 5
  Previous Assignment: None
  (New) Module Description? subtractor1 sub 16 1.0 100

> Function:  add,  Total number of nodes: 8
  Max. number of possible evaluations: 6
  Previous Assignment: None
  (New) Module Description? adder1 add 16 1.0 100

Module Selection is complete! Do you want any change (y/n)? n

Stage Latch Information:
  Dss (set-up time in nsec.) ? 10
  Dsp (propagation time in nsec.) ? 10
  Unit cost (per bit) ? 0.005

Design-space boundaries computation started ..

****************************************
*  Design-Space Boundary Information  *
****************************************

>> Fastest Design:

Nodes-to-stages Assignment:
   Stage 0: 0(sub1) 7(add1) 8(add2) 15(D1) 17(D3) 18(D4)
   Stage 1: 1(sub2) 2(sub3) 3(sub4) 9(add3) 10(add4) 16(D2) 23(J4)
   Stage 2: 4(sub5) 11(add5) 12(add6) 21(J2) 22(J3)
   Stage 3: 5(sub6) 19(D5) 20(J1)
   Stage 4: 6(sub7) 13(add7) 14(add8) 24(J5)
Initiation Interval (nsec.): 120
   Clock Cycle (nsec.): 120,    Latency: 1
Effective Initiation Interval: 120
   Resynchronization rate: 0 %
Total Cost: 17.88
   Module cost: 15.0
   # of modules: (subtractor1: 7, adder1: 8)
   Latch cost : 2.88
Scheduling Algorithm Used: Forward-Maximum-Scheduling

>> Cheapest design:
```

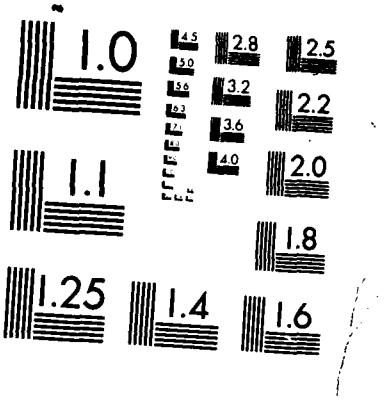
```
Nodes-to-stages Assignment:
   Stage 0:  16(D2) <2(sub3) 1(sub2)> 17(D3) 15(D1) 7(add1)
   Stage 1:  18(D4) 8(add2) 0(sub1)
   Stage 2:  23(J4) 3(sub4) 10(add4)
   Stage 3:  19(D5) 20(J1) 21(J2) <4(sub5) 5(sub6)> 22(J3)
             <11(add5) 9(add3) 12(add6)>
   Stage 4:  24(J5) 6(sub7) 14(add8)
   Stage 5:  13(add7)
Initiation Interval (nsec.): 1320
   Clock Cycle (nsec.): 220,    Latency: 6
Effective Initiation Interval: 1320
   Resynchronization rate: 0 %
Total Cost: 5.52
   Module cost: 2.0 (# of modules: (subtractor1: 1, adder1: 1))
   Latch cost : 3.52
Scheduling Algorithm Used: Forward-Nonoverlap-Scheduling

>> Absolute Boundaries:
     Absolute Minimum Cost: 5.52
     Absolute Minimum Initiation Interval: 120

**********************************
*       Main Synthesis Loop      *
**********************************
Select Optimization Mode (cost/speed/exit)? cost
>> Cost-Constrained-Synthesis Started!

Maximum Allowable Cost? 8
Expected Resynchronization Rate (in %)? 15

Design in progress with:
  Latency: 2, # of modules: (3 3)
  Stage times: (120 220 320 420 520)

> Tentative Solutions Found.

Nodes-to-stages Assignment:
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
   Stage 2:  21(J2) 22(J3) 4(sub5) <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  6(sub7)
```

```
   Stage 5:  24(J5) 14(add8) 13(add7)
Initiation Interval (nsec.): 240
   Clock Cycle (nsec.): 120,    Latency: 2
Effective Initiation Interval: 312.1128
   Resynchronization rate: 15 %
Total Cost: 9.2
   Module cost: 6.0 (# of modules:(subtractor1:3, adder1:3))
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling
                .
                .

Nodes-to-stages Assignment:
   Stage 0: 23(J4) 10(add4) 3(sub4) 16(D2) 18(D4) 8(add2)
            <2(sub3) 1(sub2)> 0(sub1) 17(D3) 15(D1) 7(add1)
   Stage 1: 24(J5) 14(add8) 6(sub7) 19(D5) 13(add7) 20(J1) 21(J2)
            <4(sub5) 5(sub6)> 22(J3) <11(add5) 9(add3) 12(add6)>
Initiation Interval (nsec.): 440
   Clock Cycle (nsec.): 220,    Latency: 2
Effective Initiation Interval: 440
   Resynchronization rate: 15 %
Total Cost: 7.36
   Module cost: 6.0 (# of modules: (subtractor1: 3, adder1: 3))
   Latch cost : 1.36
Scheduling Algorithm Used: Forward-Feasible-Scheduling


Design in progress with:
  Latency: 2, # of modules: (4 3)
  Stage times: (120 220 320 420 520)
                .
                .

Design in progress with:
  Latency: 3, # of modules: (2 2)
  Stage times: (120)

> Tentative Solutions Found.

Nodes-to-stages Assignment:
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
```

AD-A164 690 SYNTHESIS OF HIGH-SPEED DIGITAL SYSTEMS(U) UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES COMPUTER RESEARCH INST N PARK 08 NOV 85 CRI-85-23 ARO-20637.11-EL 3/3

UNCLASSIFIED DAAG29-83-K-0147 F/G 9/2 NL





```
   Stage 2:  21(J2) 22(J3) 4(sub5) <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  14(add8)
   Stage 5:  24(J5) 6(sub7) 13(add7)
Initiation Interval (nsec.): 360
   Clock Cycle (nsec.): 120,    Latency: 3
Effective Initiation Interval: 414.1116
   Resynchronization rate: 15 %
Total Cost: 7.2
   Module cost: 4.0 (# of modules: (subtractor1: 2, adder1: 2))
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling
                    .
                    .

Design in progress with:
  Latency: 3, # of modules: (3 2)
  Stage times: (120)
                    .
                    .

  *** SOLUTION ***

Nodes-to-stages Assignment:
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
   Stage 2:  21(J2) 22(J3) 4(sub5) <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  14(add8)
   Stage 5:  24(J5) 6(sub7) 13(add7)
Initiation Interval (nsec.): 360
   Clock Cycle (nsec.): 120,    Latency: 3
Effective Initiation Interval: 414.1116
   Resynchronization rate: 15 %
Total Cost: 7.2
   Module cost: 4.0 (# of modules: (subtractor1: 2, adder1: 2))
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling

** Second Alternative:

Nodes-to-stages Assignment:
```

```
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
   Stage 2:  21(J2) 22(J3) 4(sub5) <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  6(sub7)
   Stage 5:  24(J5) 14(add8) 13(add7)
Initiation Interval (nsec.): 240
   Clock Cycle (nsec.): 120,    Latency: 2
Effective Initiation Interval: 312.1128
   Resynchronization rate: 15 %
Total Cost: 9.2
   Module cost: 6.0 (# of modules:(subtractor1:3, adder1:3))
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling
          ***          ***          ***          ***


Select Optimization Mode (cost/speed/exit)? speed
>> Performance-Constrained-Synthesis Started!

Maximum Effective Initiation Interval? 400
Expected Resynchronization Rate (in %)? 20

Design in progress with:
  Latency: 3, # of modules: (2 2)
  Stage times: (120)

> Tentative Solutions Found.

Nodes-to-stages Assignment:
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
   Stage 2:  21(J2) 22(J3) 4(sub5) <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  14(add8)
   Stage 5:  24(J5) 6(sub7) 13(add7)
Initiation Interval (nsec.): 360
   Clock Cycle (nsec.): 120,    Latency: 3
Effective Initiation Interval: 432.1476
   Resynchronization rate: 20 %
Total Cost: 7.2
   Module cost: 4.0 (# of modules:(subtractor1:2, adder1:2))
```

```
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling


Design in progress with:
  Latency: 2, # of modules: (3 3)
  Stage times: (120)

> Tentative Solutions Found.

Nodes-to-stages Assignment:
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
   Stage 2:  21(J2) 22(J3) 4(sub5)
             <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  6(sub7)
   Stage 5:  24(J5) 14(add8) 13(add7)
Initiation Interval (nsec.): 240
   Clock Cycle (nsec.): 120,    Latency: 2
Effective Initiation Interval: 336.1488
   Resynchronization rate: 20 %
Total Cost: 9.2
   Module cost: 6.0 (# of modules:(subtractor1:3, adder1:3))
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling


Design in progress with:
  Latency: 3, # of modules: (2 2)
  Stage times: (120 220)
                    .
                    .

Design in progress with:
  Latency: 3, # of modules: (3 2)
  Stage times: (120)
                    .
                    .

   *** SOLUTION ***
```

```
Nodes-to-stages Assignment:
   Stage 0:  18(D4) 0(sub1) 8(add2) 17(D3) 15(D1) 7(add1)
   Stage 1:  23(J4) 16(D2) 3(sub4) 10(add4) <2(sub3) 1(sub2)>
   Stage 2:  21(J2) 22(J3) 4(sub5)
             <11(add5) 9(add3) 12(add6)>
   Stage 3:  19(D5) 20(J1) 5(sub6)
   Stage 4:  6(sub7)
   Stage 5:  24(J5) 14(add8) 13(add7)
Initiation Interval (nsec.): 240
   Clock Cycle (nsec.): 120,    Latency: 2
Effective Initiation Interval: 336.1488
   Resynchronization rate: 20 %
Total Cost: 9.2
   Module cost: 6.0 (# of modules:(subtractor1:3, adder1:3))
   Latch cost : 3.2
Scheduling Algorithm Used: Forward-Feasible-Scheduling
          ***          ***          ***          ***

Select Optimization Mode (cost/speed/exit)? exit

 ** Solution too far away from optimum!
    Do you want to try an Exhausitive Scheduling (y/n)? y

Exhausitive Scheduling in progress ..

Sorry! There is no better schedule.
Solution List is written out to <sehwa.log>.
t
-> ^D
Goodbye
[2] ^D
```

Chapter 6
Pipeline Delay Insertion

In this chapter, we discuss an exhaustive algorithm for performance improvement of an already-existing pipeline by inserting non-operational, delay stages.

6.1. Performance Improvement by Delay Insertion

In this section, we discuss insertion of delay stages (no-operation stages) in order to increase the performance of a traditional pipeline. By a traditional pipeline, we mean that the execution schedule and the execution status of a pipeline can be represented by a reservation table (developed by Davidson [Davidson 71]). We discussed this in Chapter 3. In brief, any pipeline whose schedule and execution overlap can be represented with a reservation table has physically fixed stages. In each stage, all the modules are activated and used by only one time step of the schedule at any time. Partial sharing of a stage at the same time between overlapping time steps of different tasks is not allowed. An example of a reservation table is shown in Figure 6.1-1.

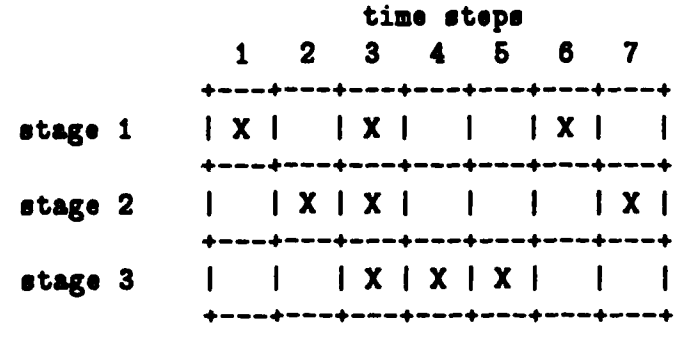


Figure 6.1-1: An example reservation table.

We assume that a fixed latency is used. In fact, as we discussed in Chapter 4, since we can achieve the minimal possible average latency by using a fixed latency scheme, there is no reason to use a variable latency scheme.

Suppose that all the rows of a reservation table are merged and represented as shown in Figure 6.1-2 -(a). Time steps which use the shared resource A are shaded. An **interval vector** which contains the distance between the time steps using the same shared resource is constructed as shown in (b). A fixed latency of 5 will cause resource conflicts for the shared resource, A, as shown in (c). Delays must be inserted into the pipe in order to avoid such conflicts, as shown in (d).

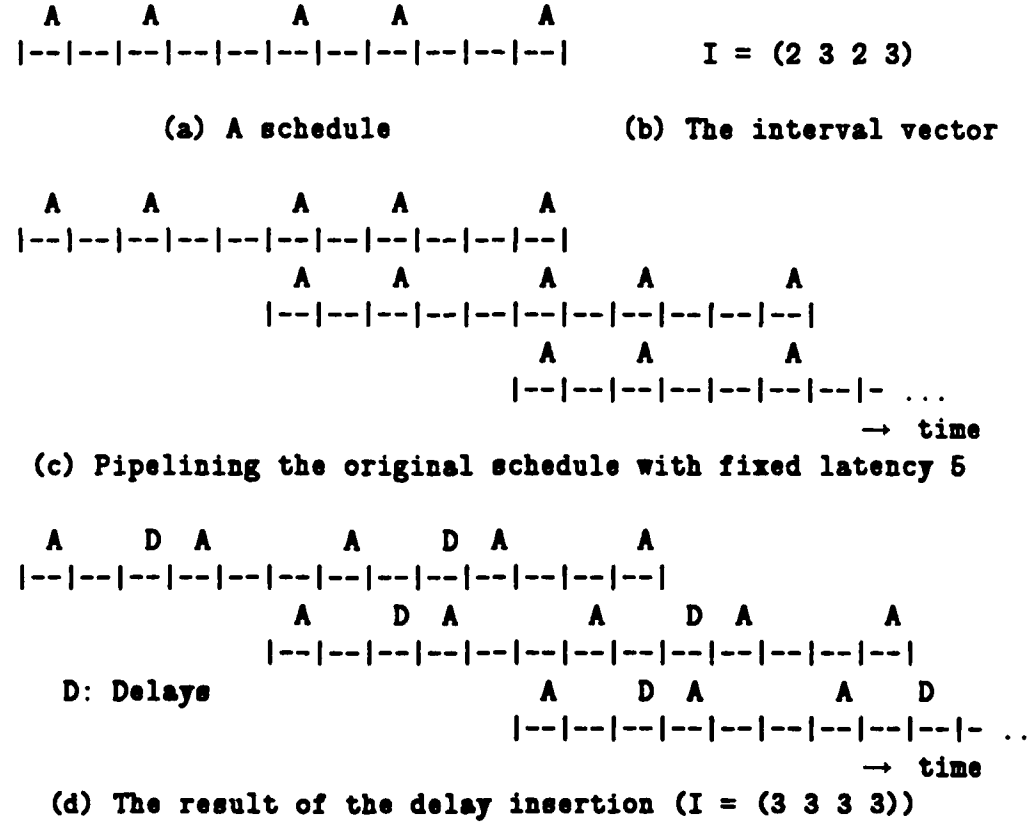


Figure 6.1-2: An example of delay insertion.

6.2. Definition of the Problem

Patel [Patel 76] has shown how the throughput of a pipeline can be improved by proper insertion of delays. In this section, we suggest a practical technique which finds an optimal delay insertion configuration for a pipeline. By "optimal delay insertion configuration", we mean that the "fixed latency" is minimized, and the number of delays to be inserted in order to achieve the minimal fixed latency is minimized.

Assuming that buffering latches for the stage latches can be added by inserting delays, we can always make the fixed latency equal to the maximum number of marks in any single row. We have already proved this with Theorem 4.3.5 and Corollary 4.3.6.

Definition 6.2.1: The **position vector** of a stage, P_s, is defined as a vector of column indices which are marked in the corresponding row, s, in the reservation table. The **interval vector** of a stage, I_s, is defined as a vector of the distances between the adjacent marks in the corresponding row, s, in the reservation table.

For example, the position and interval vectors for the reservation table in Figure 6.1-1 are $P_1 = (1\ 3\ 6)$, $P_2 = (2\ 3\ 7)$, $P_3 = (3\ 4\ 5)$, $I_1 = (2\ 3)$, $I_2 = (1\ 4)$, and $I_3 = (1\ 1)$.

Now, let us examine the conditions for a reservation table not to have any resource conflict with a certain latency.

Lemma 6.2.2: Let $P_s = (p_1\ p_2\ \ldots\ p_n)$ be the position vector for some stage, s. Also let N be the chosen fixed latency which is greater than or equal to n. If and only if $(p_j - p_i)$ is not divisible by N for every pair of i and j, $1 \leq i \leq j \leq n$, there will be no conflict in using stage s.

Proof: The time steps when stage s is used are

p_i, $p_i + N$, $p_i + 2N$, ... , $p_i + kN$, ... (due to p_i) and

p_j, $p_j + N$, $p_j + 2N$, ... , $p_j + k'N$, ... (due to p_j).

For any k and k', $(p_j + k'N) - (p_i + kN) = (k' - k)N + (p_j - p_i)$. Therefore if and only if $(p_j - p_i)$ is not divisible by N, then $(p_j + k'N) - (p_i + kN)$ cannot be zero for any k and k', and hence, $(p_j + k'N)$ and $(p_i + kN)$ cannot occur during the same time slot. Thus, if this is true for every possible pair of i and j, $1 \leq i \leq j \leq n$, then there will be no conflict in using stage s. □

Lemma 6.2.2 describes the resource-conflict-free condition in terms of the position vectors. We can also specify the resource-conflict-free condition in terms of the interval vectors.

Corollary 6.2.3: Let $I_s = (i_1\ i_2\ \ldots\ i_{n-1})$ be the interval vector for some stage, s. If and only if $\sum_{j=u}^{v} i_j$ is not divisible by N (N > n-1) for every pair of u and v, $1 \leq u \leq v \leq n-1$, then there will be no conflict in using the stage s.

Proof: For any pair of u and v, $1 \leq u \leq v \leq n-1$,

$$\begin{aligned}\sum_{j=u}^{v} i_j &= i_u + i_{u+1} + \ldots + i_v \\ &= (p_{u+1} - p_u) + (p_{u+2} - p_{u+1}) + \ldots + (p_{v+1} - p_v) \\ &= p_{v+1} - p_u\end{aligned}$$

Therefore, the rest of the proof is the same as that of Lemma 6.2.2. □

According to Lemma 6.2.2, and Corollary 6.2.3, the problem of optimal delay insertion for each row in the reservation table can be stated as follows.

For a given interval vector, $I_s = (i_1\ i_2\ i_3\ \ldots\ i_{n-1})$, where n is the minimal possible fixed latency, find any or all **delay vectors**, $K = (k_1\ k_2\ k_3\ \ldots\ k_{n-1})$, such that

$$\sum_{j=u}^{v} (i_j+k_j) \bmod n \neq 0, \text{ for any } u \text{ and } v, 1 \leq u \leq v \leq (n-1),$$

while minimizing $\sum_{j=1}^{n-1} k_j$.

The worst-case complexity of finding all possible such delay vectors is $O(n\text{-}1^n)$, since each element of K, k_i, can have any value in between 0 and n-1 (modulo n arithmetic) and there are n-1 elements in K.

However, the result of the delay insertion in a row affects the delay insertion in other rows. We illustrate this using the example reservation table of Figure 6.2-1. In this reservation table, the locations where delays can be inserted are marked with delay elements, k. (Note that delaying the mark in row 1 of column 1 does not change any interval.)

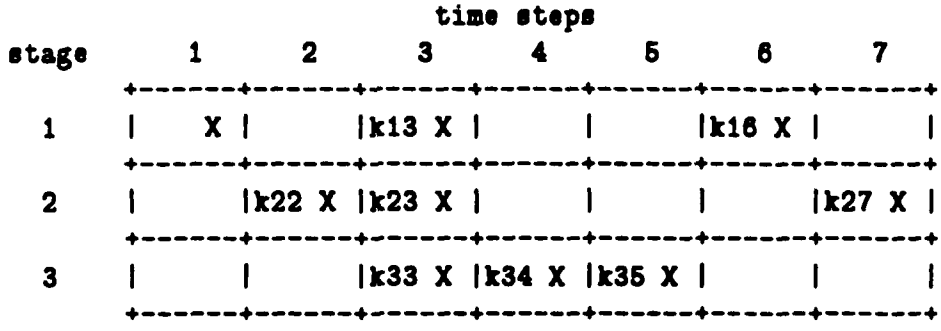


Figure 6.2-1: A reservation table marked with delay elements.

In a reservation table, data precedence relations between the computations of different stages and time steps cannot be shown. Therefore, we must assume that the computations of a time step are data dependent on every computation in the previous time step [Patel 76].

Let Ij=(ij1 ij2) be the interval vector for the j-th row. Then, for the reservation of Figure 6.2-1, I1=(2 3), I2=(1 4), and I3=(1 1). Let k*lm* be the number of delay steps to be inserted before the mark in row *l* and column *m*. If

either k13, k23 or k33 is 1, then the mark in column 4 of row 3 is delayed by one time step. Therefore, all the marks beyond column 4 must be delayed too. A modified reservation table after delay insertion with k13=2 and k23=1 is shown in Figure 6.2-2.

	time steps								
stage	1	2	3	3a	3b	4	5	6	7
1	X		D	D	X			X	
2		X	D	X					X
3			X			X	X		

* Time steps 3a and 3b are inserted as the result of delay insertion.
* D's represent the delays.

Figure 6.2-2: Delay insertion for the reservation table of Figure 6.2-1.

For the reservation table of Figure 6.2-1, the interval vectors after delay insertion will be:

$$I1 = [\{2 + k22 + k13\} \; \{3 + (\max(k13\ k23\ k33) - k13) + k34 + k35 + k16\}]$$

$$I2 = [\{1 + k23\} \; \{4 + (\max(k13\ k23\ k33) - k23) + k34 + k35 + k16 + k27\}]$$

$$I3 = [\{1 + \max(k13\ k23) + k34\} \; \{1 + k35\}]$$

and the number of time steps (or columns in the reservation table) is increased by

$$\{k22 + \max(k13\ k23\ k33) + k34 + k35 + k16 + k27\}.$$

In general, the total number of increased time steps for a reservation table which originally has n time steps is

$$\sum_{i=1}^{n} max_j\{kij\}.$$

Based on this discussion, we can state the delay-insertion problem for a rservation table with multiple rows as

Find a configuration of kij's (called a **delay vector**) such that

1. the new interval vector Ii after delay insertion, for every i, satisfies the condition of Corollary 6.2.3 with the minimum possible latency,[23] and

2. also minimizes $\sum_{i=1}^{n} max_j\{kij\}$.

6.3. An Optimal Delay Insertion Algorithm

In this section, we discuss a branch-and-bound algorithm which finds all delay vectors, K, that satisfy the conditions stated in the problem statement in the previous section. We only outline the algorithm here. A detailed description of the algorithm is in [Park 85a].

We first outline the algorithm, and then analyze some important steps of the algorithm.

Algorithm DINS

{*There are M marks in the reservation table. The minimum*}

{*possible latency is *l*. Let $D_{total} = \sum_{i=1}^{N} max_j\{kij\}$.*}

[23]The minimum possible latency is the same as the maximum number of marks in a single row, according to Theorem 4.3.5 and Corollary 4.3.6.

1. Form a cascaded counter of M stages each of which is a modulo-l counter.[24] Each stage corresponds to an element, k_{ij}, of the delay vector, K;

2. Compute a good upper bound, UB, on the number of increased time steps, i.e., D_{total}, using heuristic procedures;

{**Branch-and-Bound Search for a Solution**}

3. Increment the counter and compute the new interval vector for each row;

4. IF ($D_{total} \leq UB$)

   A. THEN

      a. IF (all the interval vectors satisfy
         the condition of Corollary 6.2.3)
      THEN
         IF ($D_{total} < UB$)
            THEN
               replace current solution set
               with this new solution;
               UB := D_{total};
            ELSE {* $D_{total} =$ UB *}
               append this solution to the
               solution set;

      b. increment the counter;

   B. ELSE D{* $D_{total} >$ UB *}
      skip the count until
      new $D_{total} \leq$ UB;

5. Repeat Step 4 until the counter overflows;

In Step 2, two quick solutions are produced in order to get a good upper

---

[24] As shown with Corollary 6.2.3, the resource conflict checking is done in modulo-l arithmetic.

bound on the number of time steps in the resulting reservation table. The first quick solution is produced by incrementing the delay elements from left to right as they appear in the reservation table. Starting from column 2, delay elements in the current column are incremented until there are no intervals which can be divided by the chosen latency up to the current time step. An example of this type of sequential adjustment is illustrated in Figure 6.3-1. Another quick solution is found by incrementing the delay elements from right to left. An example of this type of sequential adjustment is shown in Figure 6.3-1.

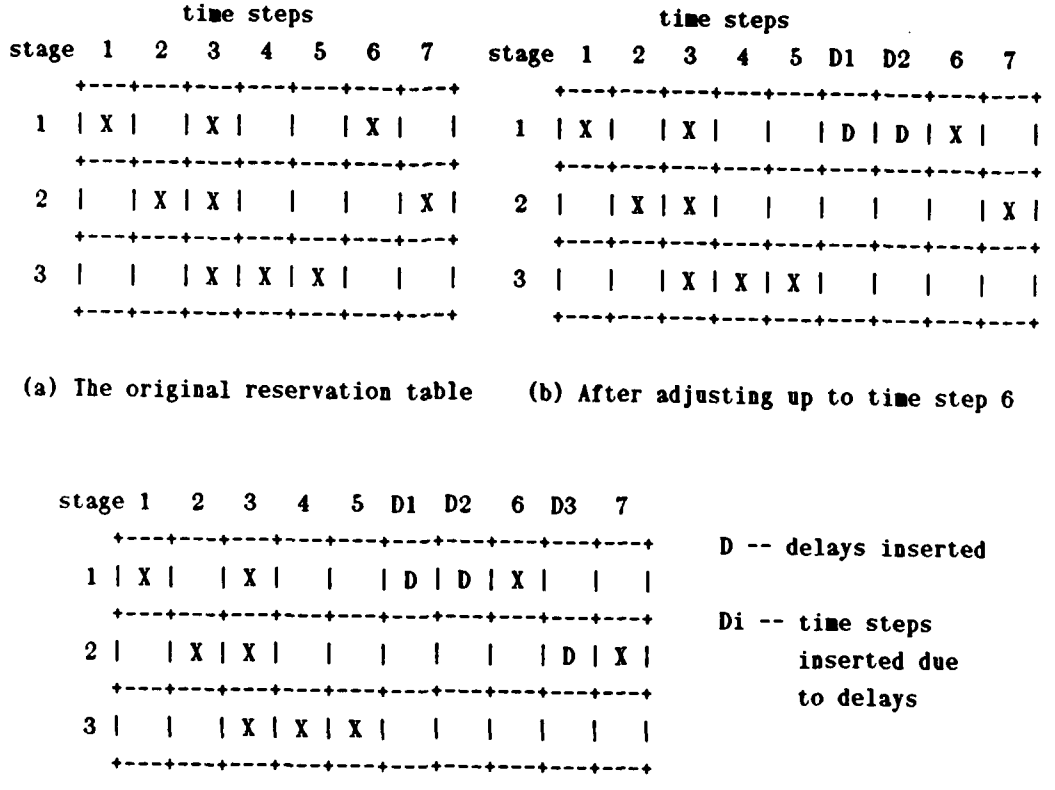


Figure 6.3-1: Left-to-right sequential adjustment.

These quick solutions are used as upper bounds on the search for the

stage	1	2	D3	D2	3	D1	4	5	6	7
1	X				X	D			X	
2		X	D	D		X				X
3						X	X	X		

Figure 6.3-2: Right-to-left sequential adjustment.

optimal solution, since a more optimal solution would be found with lower increments on the intervals. In the above example, the upper bound found by the quick solutions is three delay time steps, as shown in Figures 6.3-1 and 6.3-2.

The delay vector is incremented from right-to-left. Whenever the increment on a digit becomes latency - 1, the digit is reset to its original value and the next digit to the left is incremented. In Step 4-B, we skip some counts which are not necessary to check. Whenever D_{total} reaches the upper bound, the digit just incremented is reset to zero and the next digit to the left is incremented (note that when a digit is incremented, all the digits to the right are already reset to zero), since incrementing the digit any further will always result in a equal or larger number of delay steps. Then, the next digit to the left is incremented and the new interval vector is examined.

Figure 6.3-3 shows an optimal delay insertion for the reservation table of Figure 6.2-1. With this new reservation table, a fixed latency of 3 is possible. The number of time steps is 9, which is less than the heuristic solutions by one.

The algorithm DINS, described in the previous section, basically checks the feasibility of every possible configuration of the delay vector K, and

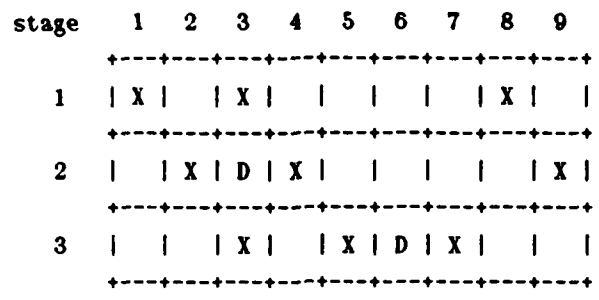


Figure 6.3-3: An optimal delay insertion for the reservation table of Figure 6.2-1 for latency 3.

compares the results. Therefore the algorithm guarantees completeness (finds all possible solutions) and correctness (every solution has the minimal number of delays).

The run-time complexity of this algorithm is as follows. Let l be the maximum number of marks in a row and r be the number of rows in the reservation table. Then, there are at most $r \cdot l$ marks in the reservation table. Thus, there are at most $(r \cdot l)^l$ different configurations of the delay vector. For each configuration of the delays, we have to check the interval vectors according to Corollary 6.2.3. For an interval vector of size l-1,[25] there are $\frac{(l-1)(l-2)}{2}$ combinations of intervals which must be checked. Therefore, the run-time complexity of the algorithm is $O(l^2 \cdot (r \cdot l)^l)$. If we consider the number of rows a constant, the run-time complexity of the algorithm is $O(l^{l+2})$.

Although the complexity of this search in time is exponential in the worst case, the bounded procedure converges very quickly. When r=3 and l=10, which is quite a complex pipe, the Franz LISP procedure converges in a few minutes of CPU time on a VAX 11/750.

---

[25] Note that the size of the interval vector of a row with l marks is l-1.

Chapter 7
Conclusions and Future Research

In this chapter the main contributions of this thesis are summarized and directions for future research in this area are discussed.

7.1. Main Contributions

The results of this research have allowed us to draw the following general conclusions:

- The concepts presented in the dissertation have been validated by means of theorems and software.
- The theoretical results are valuable because they
    - o provide a method for computing bounds which helps limit the search space during synthesis, and
    - o provide a formal basis for the algorithms which implement the synthesis.
- Good pipelines and execution overlap hardware in general can be produced automatically.
- Exhaustive algorithms can be useful for problems of practical size, even if the problem is NP-complete, if proper bounding of the search space is performed.

The main results of Chapters 2 and 3 provide a formal basis for a theory of clocking, and verify that the clocking scheme itself plays a major role in improving performance. It is also shown that there is a possibility of extending the theory to more general designs. In Chapter 3, it is shown that systolic

arrays can be completed or modified to improve performance using techniques described in the thesis.

The theory of scheduling and resource allocation in Chapter 4 provides a formal basis for pipeline synthesis at the functional level. The data flow graph model introduced in Chapter 4 allows modeling conditional branches in a pipeline. This makes it possible to use a pipeline design style in more general designs such as computer CPUs. In Chapter 5, the allocation table is introduced. The usage of the allocation table allows concurrent scheduling and resource allocation with cost and performance constraints. It is shown with examples that the allocation table is a useful structure for pipeline synthesis, comparable to a reservation table in terms of utility. It is also verified that urgency scheduling is a valid technique for pipeline synthesis.

7.2. Future Research

Future research plans include

- the inclusion of detailed interconnections, multiplexers and buses in the pipeline synthesis procedure,
- the use of an area estimator in order to make the synthesis process closer to optimal,
- the automatic design of a large problem, such as the IBM 360 using the programs described in this thesis,
- automatic synthesis of systolic arrays, and
- the use of an expert system to control execution of the various packages described in this thesis.

The first two problems need to be solved in order for the current implementation of the theory to produce designs more closer to optimal.

We now describe the systolic array problem in more detail. A number of systolic array designs have been done manually, and there is a growing consensus that this type of design is useful for a wide range of numeric and symbolic processing applications. The approach to systolic array synthesis will be to modify the pipeline synthesis package of routines to accommodate systolic arrays. The pipeline synthesis package takes as input a data flow graph with some information about required input and output timing. The same input will be used for systolic synthesis. The first step will be to detect how and where the design can be transformed into a systolic design. This is more complex than identifying pipelining opportunities, but involves the detection of feedback loops and reused values in the data flow specification. Then, the existing synthesis algorithms should be extended to include the systolic style by modifying the key information used in pipeline synthesis, the allocation table. Finally, the systolic synthesis package should be tested on example designs to insure that the software is operating properly, and to demonstrate results of the project.

Another problem to be considered is the integration of the techniques developed in this thesis into a general synthesis system. In a large system, there are many subsystems with different cost and performance requirements. For optimal usage of the techniques developed in this thesis for large designs, we need procedures that determine

- which parts of the system being implemented are either to be pipelined or to use maximum execution overlap during system partitioning and design style selection, and

- in which order the various synthesis procedures are to be executed for optimal design space exploration.

As we mentioned in Chapter 1, these high-level decision processes are computationally intractable when optimal results are desired. In order to make these decisions efficiently, design experience and intelligent reasoning

capabilities are essential. Accordingly, an expert system which is specialized in high-speed designs would be a natural choice for these tasks.

Appendix A

A Stage Partitioning Algorithm for Maximal Execution Overlap

The following algorithm, KPART, is the detailed description of the stage partitioning algorithm we outlined in Section 3.2.1.

```
Algorithm KPART (MCGs, Lmax, Dss, Dsp);

{*   MCGs    Input micro-cycle graphs                    *}
{*   Lmax    The maximum stage time limit                *}
{* Dss, Dsp  Stage latch set-up and propagation delays   *}

variable

   H    : Set of the starting nodes (modules) for the current stage
  SF    : Set of the current searching front nodes
  EH    : Set of the candidate edges for stage latches
 TEMP   : Set of the nodes to be added to current H
  NH    : Set. Starting vertices for the next H
 w(i)   : Critical path delay from the previous partition
          line to vertex i inclusively
OE(i)   : Set of all the edges coming out of vertex i
IE(i)   : Set of all the edges going into vertex i
mark(i): Boolean. True if node i has already been checked

cutset : Sets of edges for stage latches
  d    : Resulting stage propagation delays
  K    : The number of stages determined


begin{*KPART*}

for all MCGs in parallel do
```

```
NH := {root(s)}; {* wave front starts from root nodes *}
EH := { };   K := 1;   dmax := 0;
w(i) := δ_i for every i ∈ NH;

{* repeat until there is no node left *}
repeat {* until empty(NH) *}

   {* get starting nodes in TEMP for a new partition  *}
   K := K + 1;   TEMP := NH;   NH := {};

   {* initialize the propagation delays *}
   if (k > 1) then
       for every i ∈ TEMP do w(i) := δ_i + Dsp;

   {* get a stage *}
   repeat {* until empty(TEMP) *}

      H := TEMP;     TEMP := {};

      {* remove all indirect vertices *}
      for every i ∈ H do H := H - descendents(i);

      {* get searching fronts in SF *}
      SF := {};
      for every i ∈ H do
         SF := SF + children(i);
         {* get candidate edges for stage latch in EH *}
         EH := EH + OE(i);
      for every j ∈ SF do SF := SF - descendents(j);
      for every j ∈ SF do mark(j) := false;

      for every i ∈ H do
         for every j ∈ children(i) do
            if j ∈ SF then
               if w(i) + δ_j + Dss > Lmax
                  then
                     {* if including child j exceeds stage *}
                     {* time limit,   put stage latches on *}
                     {* all edges going into child j       *}
                     cutset(k) := IE(j);
                     EH := EH - IE(j);
```

```
                    {* update stage propagation delays    *}
                    d(K) := w(i) + Dss;
                    if d(K) > dmax then dmax := d(K);

                    {* if j is not a leaf,  put it in NH  *}
                    if j not a leaf then NH := NH + j;

                    {* if j in TEMP already, remove it    *}
                    if mark(j) then TEMP := TEMP - j;

                 else if j not a leaf then
                    {* move searching head *}
                    if not mark(j)
                       then
                          mark(j) := true;
                          TEMP := TEMP + j;
                          w(j) := w(i) + δj;
                          {* update candidate edges for  *}
                          {* stage latches               *}
                          EH := EH - IE(j) + OE(j);

                       {* if child j is already in TEMP, *}
                       {* update its critical path delay *}
                       else if w(j) < w(i) + δj then
                          w(j) := w(i) + δj;

          until empty(TEMP)

          {* No more nodes can be added due to stage time limit.*}
          {* Put all the edges still remaining in the cutset.   *}
          {* These edges go into vertices beyond the current SF.*}
          cutset(k) := cutset(k) + EH;

       until empty(NH);

    end{*KPART*}
```

Run Time Analysis

Let |E| be the total number of edges in the input micro-cycle graphs.

Each node can be in SF only once and in NH at most once. For each node in either SF or NH, every output edge is traversed once and a child node is checked once. Therefore, the run-time complexity of this algorithm is O(|E|).

An Example Trace of the Partitioning Procedure

The following is a step-by-step trace of the stage partitioning of the micro-cycle graph of Figure 3.2-1.

Algorithm KPART(G, Lmax=85, Dss=5, Dsp=10);

1. Initially, cutset(1) = $e_{0,1}$ (**the first stage latch L1**), H = $\{v_1\}$, SF = $\{v_2\}$, and EH = $\{e_{1,2}\}$.
2. v_2 can be included in the first partition since $\delta_1 + \delta_2 + Dss \leq Lmax$. Thus H is updated and new SF is computed.
   a. v_2 is put in TEMP and moved to H. TEMP is cleared.
   b. SF gets $\{v_3, v_4, v_5, v_6, v_{12}\}$.
   c. EH becomes $\{e_{2,3}, e_{2,4}, e_{2,5}, e_{2,6}, e_{2,12}\}$.
   d. Vertices v_3, v_4, v_5, and v_6 are removed from SF since they are descendents of v_{12}.
3. v_{12} in SF cannot be included in the first partition since ($\delta_1 + \delta_2 + \delta_{12} + Dss$) exceeds Lmax.
   a. EH = EH - IE(v_{12}). $e_{2,12}$ is removed from EH and put in cutset(2).
   b. v_{12} is put in NH to become a head for the second stage.
   c. d(1) and dmax are updated with ($\delta_1 + \delta_2 + Dss$) = 85 .
4. TEMP is empty. Thus, all the edges in EH are also put in cutset(2).

The locations for the second stage latches are $e_{2,3}$, $e_{2,4}$, $e_{2,5}$, $e_{2,6}$, and $e_{2,12}$. {**the second stage latches(L2)**}.

5. v_{12} is moved from NH to H and new SF and EH are computed.
   a. $w(12) = \delta_{12} + Dsp = 25$, $SF = \{v_3, v_4, v_5, v_6\}$.
   b. $EH = \{e_{2,3}, e_{2,4}, e_{2,5}, e_{2,6}\} + \{e_{12,3}, e_{12,4}, e_{12,5}, e_{12,6}\}$.
6. All the current searching-front vertices in SF can be included in the second stage. Thus TEMP is updated to contain v_3, v_4, v_5, and v_6. The corresponding updating procedures during the initialization pass of the inner "repeat" loop are:
   a. $H = \{v_3, v_4, v_5, v_6\}$, $w(3) = w(6) = 45$, $w(4) = w(5) = 40$.
   b. $EH = \{e_{3,7}, e_{4,8}, e_{5,9}, e_{6,10}\}$.
   c. $SF = \{v_7, v_8, v_9, v_{10}\}$ - **descendents**$(v_7) = v_7$.
7. v_7 can be included in the second stage and thus v_7 becomes the next searching head.
   a. $H=\{v_7\}$ $(w(7) = 70)$, $EH=\{e_{7,8}, e_{4,8}, e_{5,9}, e_{6,10}\}$, $SF = \{v_8\}$.
8. Including v_8 violates the maximum stage propagation delay $(w(7) + \delta_8 + Dss = 140)$.
   a. $NH = \{v_8\}$, $cutset(3) = \{e_{7,8}, e_{4,8}, e_{5,9}, e_{6,10}\}$ {**the third stage latches(L3)**}.
   b. $d(2) = w(7) + Dss = 75$, $EH = EH - IE(v_8) = \{e_{5,9}, e_{6,10}\}$.
9. $H = \{v_8\}$, $w(8) = \delta(8) + Dsp = 75$, $SF = \{v_9\}$.

   $EH = EH + \{e_{8,9}\} = \{e_{5,9}, e_{6,10}, e_{8,9}\}$.
10. $w(8) + \delta_9 + Dss = 100 > Lmax$. Thus

    $cutset(4) = EH = \{e_{5,9}, e_{6,10}, e_{8,9}\}$ {**the fourth stage latches(L4)**}.

    $NH = \{v_9\}$, $w(9) = 30$, $SF = \{v_{10}\}$.

$EH = EH - IE(v_9) + OE(v_9) = \{e_{6,10}, e_{9,10}\}$.

11. The remaining vertices, v_9 and v_{10} becomes the fourth stage and are terminated by **the fifth stage latch(L5)**. $d(4) = 40$.

12. Finally, d(5) is determined by the storage propagation delay of L5.

Appendix B
A Node Coloring Algorithm

The following algorithm, DJ-Coloring, colors the nodes of a data flow graph as discussed in Section 4.5.

Algorithm DJ-Coloring;

BEGIN {*coloring*}

PHASE I: Initialization

1. Set color_code to 0; {*initialize color palette*}

2. Get all the root nodes, {roots}.[26]

3. FOR EVERY node in {root} DO
   set color[node] := color_code;
   increment color_code;

4. Put all the root nodes in the coloring wave front, WF, as the starting wave front.

PHASE II: Color Wave Propagation

5. {* Propagate color one level *}
   FOR EVERY node in WF DO
   BEGIN {*node*}

      FOR EVERY children of node DO
      BEGIN {*child*}

[26] Nodes without any parents.

```
CASE node OF
BEGIN {*case*}

  DISTRIBUTE:
    if (child is a join)
      then
        color[child] := color[node];
        {*copy node's color*}
      else
        color[child] := color[node] + child#;
        {*copy node's and add one more digit*}

  JOIN:
    if (node has a single digit color)
      then
        {*when return to non-conditional*}
        {*execution path, change color*}
        color[node] := color_code;
        increment color code;

    if (child is a join) and (previous color of
         child has less # of digits than node)
      then
        color[child] := color[node] - child#;
        {*remove the last digit from node's*}
      else if (previous color of child has
              less # of digits than node's)
        color[child] := color[node];

  ELSE:
    if (child is a join) and (previous color of
         child has less # of digits than node)
      then
        color[child] := color[node] - child#;
        {*remove the last digit from node's*}
      else if (previous color of child has
              less # of digits than node's)
        color[child] := color[node] + child#

  END {*case*}

END {*child*}

END {*node*}
```

6. {*Get new wave front nodes*}
   WF := {All the children of the nodes in WF all of
         whose parents are already colored}

7. IF (NOT EMPTY WF) GOTO 5.

END {*coloring*}

Runtime complexity of the algorithm and its implementation in LISP is $O[n^2]$ for any arbitrary graph, where n is the number of nodes in the graph. Each node can be in the wave front, WF, only once. For each node in WF, there can be at most n children, and thus the inner loop iterates at most n times for each node in WF. One iteration of the inner loop (the CASE block) takes constant number of steps. Therefore, the time complexity is $O[n^2]$. However, if we limit the fanout of a node, each node can have only a constant number of children. This makes the inner loop iterate only a constant number of times, which makes the time complexity of the algorithm linear or O[n].

Appendix C

An Algorithm for Counting Module-usage Frequency

The algorithm shown below, MinModFreq, which is based on the node coloring technique we discussed in Section 4.6, counts the minimum number of times a certain type of modules must be used to execute a part or all of the data flow graph. We assume that we know which type of operations are to be performed by which type of modules. For example, all the 12-bit additions and 16-bit additions are to be performed by 16-bit carry save adders.

Algorithm MinModFreq (Graph, Module-type)

```
; Count the minimum required number of times of usage
; of Module-type module to execute the graph, Graph.

BEGIN {*MinModFreq*}

   {* initialization *}
1. FOR every node in the graph DO
      parent_cnt(node) := 0 ; the # of already checked parents
      count(node) := 0      ; current module count up to node
      NWF := {}             ; next checking wave front nodes
      sum := 0              ; the total module count

   {* Check and count nodes of one level *}
2. FOR every node in CWF DO
   BEGIN {*loop*}
```

```
{* Get the nodes all whose parents are  *}
{* checked as the next wave front nodes.*}
FOR every child of node DO
  parent_cnt(child) := parent_cnt(child) +1;
  IF (parent_cnt(child) = #_of_parents(child))
    THEN
      NWF := NWF + {child};

{* Propagate module counter *}
CASE node OF
BEGIN {*CASE*}

  conditional:

    FOR every child of node DO
      parent_cnt(child) := parent_cnt(child) +1;
      IF (parent_cnt(child) = #_of_parents(child))
           AND  (child is not a terminal node)
        THEN
          NWF := NWF + {child};

    FOR every child of node DO
      IF (child is a join)
        THEN
          count(child) := max{count(child), count(node)};
        ELSE IF (child uses Module-type)
          count(child) := count(node) +1;
        ELSE
          count(child) := count(node);

  unconditional:

    pick a child (any_child);

    IF (any_child is conditional)
      THEN
        sum := sum + count(node);
      ELSE IF (any_child uses Module-type) AND
           (count(any_child) = 0)
        count(any_child) := count(node) + 1;
      ELSE
        count(any_child) := count(node) + count(any_child);
```

```
      END {*CASE*}

    END {*loop*}

    {* If there is any next searching front nodes, put *}
    {* them in the current wave front buffer and loop. *}
3. IF NOT (empty NWF)
    THEN
      CWF := NWF;
      NWF := {};
      GOTO 2;

4. FOR every terminal_node with a single digit color code DO
      sum := sum(terminal_node);

5. Group the terminal nodes with multi-digit color codes
   according to their first digit (outermost D-J block);

   FOR each group of nodes DO
      sum := sum + max{count(node ϵ group)};

6. RETURN sum;

END {*MinModFreq*}
```

Run time complexity of this algorithm is $O[n^2]$, where n is the number of nodes in the graph. Step 3 takes a constant number of steps. Steps 1, 4, and 5 takes O[n] times steps since there are n nodes in the graph. A node can be in CWF only once and therefore the CASE block of Step 2 is iterated at most n times. Each node has at most n-1 children, thus, one iteration of the CASE block takes O[n] time steps.

References

[Agerwala 76] Agerwala, T.
Microprogram Optimization: A Survey.
IEEE Transactions on Computers C-25(10):962-973, October, 1976.

[Aho 77] Aho, A. and Ullman, J.
Principles of Compiler Design.
Addison-Wesley, Massachusets, 1977.

[Andrews 80] Andrews, M.
Principles of Firmware Engineering in Microprogram Control.
Computer Science Press, 1980.

[Baker 74] Baker, K.
Introduction to Sequencing and Scheduling.
John Wiley & Sons, 1974.

[Berg 79] Berg, H. K.
A Model of Timing Characteristics in Computer Control.
Euromicro 5, July, 1979.

[Boulaye 71] Boulaye, G.
Microprogramming.
John Wiley & Sons, New York, N.Y., 1971.

[Breuer 72] Breuer, M. (ed.).
Digital System Design Automation, vol.I: Theory and Techniques.
Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Chen 75] Chen, T. C.
Overlap and Pipeline Processing.
Introduction to Computer Architecture.
SRA, Chicago, 1975.
Editor: Stone, H. C.

[Cook 79] Cook, P., Chung, H., and Stanley, S.
A Study in the Use of PLA-Based Macros.
Solid-State Circuits SC-14:833-840, October, 1979.

[Cotten 65] Cotton, L. W.
Circuit Implementation of High-Speed Pipeline Systems.
In *Proceedings of FJCC*, pages 489-504. AFIPS, 1965.

[Darringer 69] Darringer, J.
The Description, Simulation and Implementation of Digital Computer Processors.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1969.

[Dasgupta 76] Dasgupta, S. and Tartar, J.
The identification of maximal parallelism in straight-line microprograms.
IEEE Transactions on Computers C-25(10):986-992, October, 1976.

[Davidson 71] Davidson, E.
The Design and Control of Pipelined Function Generators.
In *Proceedings of 1971 International IEEE Conference on Systems, Networks, and Computers*, pages 19-21. 1971.

[Davidson 75] Davidson, E., et al.
Effective Control for Pipelined Computers.
In *COMPCON Digest*, pages 181-184. 1975.

[Davis 82] Davis, A. and Keller, R.
Data Flow Program Graphs.
IEEE Computer 15(2):26-41, February, 1982.

[Dennis 74] Dennis, J. B.
First Version of a Data Flow Procedure Language.
Lecture Notes in Computer Science.
Springer-Verlag, 1974, pages 362-374.

[Dennis 75] Dennis, J. B. and Misunas, D. P.
A Preliminary Data Flow Architecture for a Basic Data Flow Processor.
In *2nd Symposium on Computer Architecture*, pages 126-132. 1975.

[Dervos 83] Dervos, D. and Parker, A. C.
A Technique for Automatically Producing Optimized Digital Designs.
In *Proceedings of the Mediterranean Electrotechnical Conference,Athens*, pages B2.04. IEEE, May, 1983.

[Estrin 78] Estrin, G.
A methodology for design of digital systems - supported by SARA at the age of one.
In *Proceedings of National Computer Conference*, pages 313-324. NCC, 1978.

[Foulk 80] Foulk, P. W. and O'Callaghan, J.
AIDs - an integrated design system for digital hardware.
In *IEE Proceeding Vol.127, No.2.* IEE, March, 1980.

[Friedman 69] Friedman, T. and Yang, S.
Methods Used in an Automatic Logic Design Generator (ALERT).
IEEE Transactions on Computers C-18(7):593-614, July, 1969.

[Friedman 75] Friedman, A. and Menon, P.
Theory and Deesign of Switching Circuits.
Computer Science Press, Woodland Hills, California, 1975.

[Gary 79] Gary, M. and Johnson, D.
Computers and Intractability: A guide to the Theory of NP-Completeness.
Freeman, 1979.

[Girczyc 84] Girczyc, E. F. and Knight, J. P.
An ADA to Standard Cell Hardware Compiler Based on Graph Grammers and Scheduling.
In *Proceedings of the ICCD '84.* IEEE Computer Society, October, 1984.

[Granacki 85] Granacki, J., Knapp, D., and Parker, A.
The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface.
In *Proceedings of the 22nd Design Automation Conference.* ACM-IEEE, June, 1985.

[Hafer 81] Hafer, L.
Automated Data-Memory Synthesis : A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
PhD thesis, Dept of Electrical Engineering, Carnegie Mellon University, Pittsburgh, Pa., May, 1981.

[Hafer 83] Hafer, L. and Parker, A.
A Formal Method for the Specification Analysis, and Design of Register-Transfer Level Digital Logic.
IEEE Transactions on Computer-Aided Design CAD-2(1), January, 1983.

[Hitchcock 83] Hitchcock, C. Y.
Automated Synthesis of Data Paths.
Master's thesis, Carnegie-Mellon University, 1983.

[Horowitz 78] Horowitz, E. and Sahni, S.
Fundamentals of Computer Algorithms.
Computer Science Press, 1978.

[Irani 72] Irani, K. and McClain, G.
Optimal Design of Central Processor Data Paths.
Technical Report 58, Systems Engineering Laboratory, University of Michigan, Ann Arbor, Michigan, May, 1972.

[Kartashev 82] Kartashev, S. P., Kartashev, S. I., and Vick, C. R.
Historic Progress in Architectures for Computers and Systems.
Designing and Programming Modern Computers and Systems.
Prentice-Hall, 1982, pages 1-29, Chapter 1.

[Katzan 71] Katzan, H.
Computer Organization and the System/370.
Van Norstrand Reinhold, 1971.

[Keller 75] Keller, R.
Look-Ahead Processors.
Computing Surveys (7), December, 1975.

[Knapp 83] Knapp, D. and Parker, A.
A Data Structure for VLSI Synthesis and Verification.
Technical Report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October, 1983.

[Knapp 85] Knapp, D. and Parker, A.
A Unified Represention for Design Information.
In *Proceedings of the IFIP Conference on Hardware Description Languages*. IFIP, August, 1985.

[Kogge 81] Kogge, P. M.
The Architecture of Pipelined Computers.
McGraw-Hill, New York, N.Y., 1981.

[Kowalski 83] Kowalski, T. J. and Thomas, D. E.
The VLSI Design Automation Assistant: Prototype System.
In *Proceedings of the 20th Design Automation Conference.* IEEE, 1983.

[Kurdahi 85] Kurdahi, F. and Parker, A.
Area Estimation of VLSI Integrated Circuits.
Technical Report CRI-85-05, EE-Systems Dept. USC, 1985.

[Lang 79] Lang, D. E., Agerwala, T. K., and Chandy, K. M.
A Modeling approach and design tool for pipelined central processors.
In *Proceedings of the 6th Annual Symposium on Computer Architecture.* IEEE Computer Society, April, 1979.

[Lawson 78] Lawson, G.
Design Style Selector, An Automated Computer Program Implementation.
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., August, 1978.

[Leiserson 82] Leiserson, C. E., Rose, F. M., and Saxe, J. B.
Digital circuit optimization.
Technical Report, Dept. of Electrical Engineering and Computer Science, M.I.T., 1982.

[Leiserson 83] Leiserson, C. E., Rose, F. M. and Saxe, J. B.
Optimizing synchronous circuitry by retiming.
In *Proceedings of Third Caltech Conference on VLSI*, pages 23-36. Computer Science Press, 1983.

[McKeeman 75] McKeeman, W. M.
Stack Processors.
Introduction to Computer Architecture.
SRA, Chicago, 1975.

[Nagle 80] Nagle, A.
Automatic Design of Sequencers for The Control of Digital Hardware.
PhD thesis, Carnegie-Mellon University, October, 1980.

[Park 84] Park, N. and Parker, A.
Synthesis of Optimal Clocking Schemes for Digital Systems.
Technical Report DISC/84-1, Dept. of EE-Systems, University of Southern California, May, 1984.

[Park 85a] Park, N. and Parker, A.
Synthesis of Optimal Pipeline Clocking Schemes.
Technical Report DISC/85-1, Dept. of EE-Systems, University of Southern California, January, 1985.

[Park 85b] Park, N. and Parker, A.
Synthesis of Optimal Clocking Schemes.
In *Proceedings of the 22nd Design Automation Conference.* ACM IEEE, June, 1985.

[Parker 79] Parker, A. C., et al.
The CMU Design Automation System.
In *Design Automation Conference Proceedings No. 16.* ACM SIGDA, IEEE Tech. Comm. on Design Automation, June, 1979.

[Parker 84] Parker, A., Park, N., and Knapp, D.
Simulation Effectiveness and Design Verification.
Technical Report DISC/84-2, Department of EE-Systems, University of Southern California, October, 1984.

[Patel 76] Patel, J. H. and Davidson, E. S.
Improving the Throughput of a Pipeline by Insertion of Delays.
In *IEEE/ACM 3rd Ann. Symp. on Computer Arch.*, pages 159-163. 1976.

[Patterson 76] Patterson, D.
STRUM: Structured Microprogram Development System for Correct Firmware.
IEEE Trasnactions on Computers C-25(10):974-985, October, 1976.

[Ramamoorthy 75]
Ramamoorthy, C. V. and Li, H. F.
Some Problems in Parallel and Pipeline processing.
In *Proceedings of COMPCON, IEEE*, pages 177-180. 1975.

[Ramamoorthy 77]
Ramamoorthy, C. V.
Pipeline Architecture.
Computing Surveys 9(1):61-102, March, 1977.

[Robertson 79] Robertson, E.
Microcode bit optimization is NP complete.
IEEE Transactions on Computers C-28(4):316-319, April, 1979.

[Sastry 82] Sastry, S. and Parker, A.
The Complexity of Two-Dimensional Compaction of VLSI Layouts.
In *Proceedings of the 1982 IEEE International Conference on Circuits and Computers*, pages 402-406. IEEE, September, 1982.

[Sastry 83] Sastry, S.
Wiring Space Estimation of Master Slice IC's.
Technical Report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, June, 1983.

[Shar 72] Shar, L. E.
Design and Scheduling of Statically Configured Pipelines.
Technical Report Digital Systems, Lab Report SU-SEL-72-042, Stanford University, September, 1972.

[Snow 78] Snow, E.
Automation of Module Set Independent Register Transfer Level Design.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1978.

[Thomas 77] Thomas, D.
The Design and Analysis of an Automated Design Style Selector.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1977.

[Tseng 84] Tseng, C.
Automated Synthesis of Data Paths in Digital Systems.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1984.

[Unger 83] Unger, S. and Tan, C.
Optimal Clocking Schemes for High Speed Digital Systems.
In *Proceedings of the ICCD Conference*, pages 366-369. IEEE Computer Society, October, 1983.

[Zimmermann 79]
Zimmermann, G.
The MIMOLA Design System: A Computer Aided Digital Processor Design Method.
In *Proceedings of the 16th Design Automation Conference*, pages 53-58. ACM SIGDA, IEEE Computer Society - DATC, June, 1979.

DTIC

FILMED

4-86

END